Oracle8™

Concepts

Release 8.0

December, 1997

Part No. A58227-01

ORACLE®

Enabling the Information Age™

Oracle8 Concepts

Part No. A58227-01

Release 8.0

Copyright © 1997 Oracle Corporation. All rights reserved.

Primary Author: Lefty Leverenz

Contributors: Richard Allen, David Anderson, Andre Bakker, Steve Bobrowski, Bill Bridge, Atif Chaudry, Cynthia Chin-Lee, Cindy Closkey, Jeff Cohen, Benoit Dageville, Sandy Dreskin, Jason Durbin, Ahmed Ezzat, Diana Foch-Lorentz, John Frazzini, Anurag Gupta, Gary Hallmark, Michael Hartstein, Terry Hayes, Alex Ho, Chin Hong, Ken Jacobs, Sandeep Jain, Amit Jasuja, Hakan Jakobsson, Robert Jenkins, Jr., Ashok Joshi, Jonathan Klein, R. Kleinro, Robert Kooi, Vishu Krishnamurthy, Andre Kruglikov, Tirthankar Lahiri, Juan Loaiza, Brom Mahbod, Richard Mateosian, William Maimone, Andrew Mendelsohn, Reza Monajjemi, Mark Moore, Rita Moran, Denise Oertel, Mark Porter, Maria Pratt, Tuomas Pystynen, Patrick Ritto, Hasan Rizvi, Sriram Samu, Hari Sankar, Gordon Smith, Danny Sokolsky, Leng Leng Tan, Lynne Thieme, Alvin To, Alex Tsukerman, William Waddington, Joyo Wijaya, Linda Willis, Andrew Witkowski, Mohamed Zait

Graphic Designer: Valarie Moore

The programs are not intended for use in any nuclear, aviation, mass transit, medical, or other inherently dangerous applications. It shall be licensee's responsibility to take all appropriate fail-safe, back up, redundancy and other measures to ensure the safe use of such applications if the Programs are used for such purposes, and Oracle disclaims liability for any damages caused by such use of the Programs.

This Program 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 patent and other intellectual property law. Reverse engineering of the software is prohibited.

The information contained 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.

If this Program 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 Programs delivered subject to the DOD FAR Supplement are 'commercial computer software' and use, duplication and disclosure of the Programs shall be subject to the licensing restrictions set forth in the applicable Oracle license agreement. Otherwise, Programs delivered subject to the Federal Acquisition Regulations are 'restricted computer software' and use, duplication and disclosure of the Programs shall be subject to the restrictions in FAR 52..227-14, Rights in Data -- General, including Alternate III (June 1987). Oracle Corporation, 500 Oracle Parkway, Redwood City, CA 94065.

Oracle SQL*Loader, SQL*Net and SQL*Plus are registered trademarks of Oracle Corporation, Redwood City, California.

Net8, Oracle Call Interface, Oracle7, Oracle8, Oracle Forms, Oracle Enterprise Manager, Oracle Parallel Server, Oracle Server Manager, PL/SQL, Pro*C, Pro*C/C++, and Trusted Oracle are trademarks of Oracle Corporation, Redwood City, California.

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

Contents

Send Us Your Comments xxiii

Preface xxv

Part I What Is Oracle?

1 Introduction to the Oracle Server

Databases and Information Management 1-2
The Oracle Server 1-4
Oracle Databases 1-8
Database Structure and Space Management 1-8
Logical Database Structures 1-8
Physical Database Structures 1-11
Memory Structure and Processes 1-13
Memory Structures 1-13
Process Architecture 1-16
The Program Interface 1-19
An Example of How Oracle Works 1-19
Data Concurrency and Consistency 1-20
Concurrency 1-20
Read Consistency 1-21
Locking Mechanisms 1-22
Distributed Processing and Distributed Databases 1-23
Client/Server Architecture: Distributed Processing 1-23

Distributed Databases 1-24
Table Replication 1-26
Oracle and Net8 1-26
Startup and Shutdown Operations 1-27
Database Security 1-27
Security Mechanisms 1-28
Trusted Oracle 1-34
Database Backup and Recovery 1-34
Why Is Recovery Important? 1-34
Types of Failures 1-35
Structures Used for Recovery 1-37
Basic Recovery Steps 1-39
The Recovery Manager 1-40
The Object-Relational Model for Database Management 1-40
The Relational Model 1-41
The Object-Relational Model 1-41
Schemas and Schema Objects 1-42
The Data Dictionary 1-47
Data Access 1-48
SQL — The Structured Query Language 1-48
Transactions 1-49
PL/SQL 1-52
Data Integrity 1-54

Part II Database Structures

2 Data Blocks, Extents, and Segments

The Relationships Among Data Blocks, Extents, and Segments 2-2
Data Blocks 2-3
Data Block Format 2-3
An Introduction to PCTFREE, PCTUSED, and Row Chaining 2-5
Extents 2-10
When Extents Are Allocated 2-11
Determining the Number and Size of Extents 2-11

How Extents Are Allocated ... 2-11
When Extents Are Deallocated ... 2-13
Segments ... 2-15
Data Segments ... 2-15
Index Segments ... 2-15
Temporary Segments ... 2-16
Rollback Segments ... 2-17

3 Tablespaces and Datafiles

An Introduction to Tablespaces and Datafiles ... 3-2
Tablespaces ... 3-3
The SYSTEM Tablespace ... 3-4
Allocating More Space for a Database ... 3-4
Bringing Tablespaces Online and Offline ... 3-7
Read-Only Tablespaces ... 3-9
Temporary Tablespaces ... 3-10
Datafiles ... 3-11
Datafile Contents ... 3-12
Size of Datafiles ... 3-12
Offline Datafiles ... 3-12

4 The Data Dictionary

An Introduction to the Data Dictionary ... 4-2
The Structure of the Data Dictionary ... 4-2
SYS, the Owner of the Data Dictionary ... 4-3
How the Data Dictionary Is Used ... 4-3
How Oracle Uses the Data Dictionary ... 4-3
How Oracle Users Can Use the Data Dictionary ... 4-5
The Dynamic Performance Tables ... 4-7

Part III The Oracle Instance

5 Database and Instance Startup and Shutdown

Overview of an Oracle Instance 5-2
The Instance and the Database 5-2
Connecting with Administrator Privileges 5-3
Parameter Files 5-4
Instance and Database Startup 5-5
Starting an Instance 5-5
Mounting a Database 5-6
Opening a Database 5-7
Database and Instance Shutdown 5-8
Closing a Database 5-8
Dismounting a Database 5-8
Shutting Down an Instance 5-9

6 Memory Structures

Introduction to Oracle Memory Structures 6-2
System Global Area (SGA) 6-2
The Database Buffer Cache 6-3
The Redo Log Buffer 6-6
The Shared Pool 6-6
Size of the SGA 6-11
Controlling the SGA's Use of Memory 6-12
Program Global Areas (PGA) 6-13
Contents of a PGA 6-13
Size of a PGA 6-14
Sort Areas 6-15
Sort Direct Writes 6-16
Virtual Memory 6-16
Software Code Areas 6-16

7 Process Structure

Introduction to Processes 7-2
Single-Process Oracle 7-2
Multiple-Process Oracle 7-3
User Processes 7-4
Oracle Processes 7-5
Trace Files and the ALERT File 7-14
Variations in Oracle Configuration 7-16
Single-Task Configuration 7-16
Dedicated Server (Two-Task) Configuration 7-18
The Multithreaded Server 7-20
Examples of How Oracle Works 7-24
An Example of Oracle Using Dedicated Server Processes 7-25
An Example of Oracle Using the Multithreaded Server 7-26
The Program Interface 7-27
Program Interface Structure 7-27
The Program Interface Drivers 7-27
Operating System Communications Software 7-28

Part IV The Object-Relational DBMS

8 Schema Objects

Overview of Schema Objects 8-2
Tables 8-3
How Table Data Is Stored 8-4
Nulls 8-7
Default Values for Columns 8-8
Nested Tables 8-9
Views 8-10
Storage for Views 8-11
How Views Are Used 8-11
The Mechanics of Views 8-12
Dependencies and Views 8-13

Updatable Join Views ... 8-13
Object Views ... 8-14
The Sequence Generator ... 8-14
Synonyms ... 8-15
Indexes ... 8-17
Unique and Non-Unique Indexes ... 8-17
Composite Indexes ... 8-18
Indexes and Keys ... 8-19
How Indexes Are Stored ... 8-19
Reverse Key Indexes ... 8-22
Bitmap Indexes ... 8-23
Index-Organized Tables ... 8-28
Benefits of Index-Organized Tables ... 8-29
Index-Organized Tables with Row Overflow Area ... 8-29
Applications of Interest for Index-Organized Tables ... 8-30
Clusters ... 8-32
Performance Considerations ... 8-34
Format of Clustered Data Blocks ... 8-34
The Cluster Key ... 8-35
The Cluster Index ... 8-35
Hash Clusters ... 8-36
How Data Is Stored in a Hash Cluster ... 8-37
Hash Key Values ... 8-39
Hash Functions ... 8-40
Allocation of Space for a Hash Cluster ... 8-41

9 Partitioned Tables and Indexes

Introduction to Partitioning ... 9-2
What Is Partitioning? ... 9-2
Advantages of Partitioning ... 9-4
Very Large Databases (VLDBs) ... 9-4
Reducing Downtime for Scheduled Maintenance ... 9-6
Reducing Downtime Due to Data Failures ... 9-7
DSS Performance ... 9-7
I/O Performance ... 9-8

Disk Striping: Performance versus Availability ... 9-8
Partition Transparency ... 9-9
Manual Partitioning with Partition Views ... 9-10
Basic Partitioning Model ... 9-11
Range Partitioning ... 9-12
Partition Names ... 9-14
Partition Bounds and Partitioning Keys ... 9-14
Equipartitioning ... 9-18
Rules for Partitioning Tables and Indexes ... 9-21
Table Partitioning ... 9-21
Index Partitioning ... 9-22
DML Partition Locks ... 9-30
Performance Considerations for Oracle Parallel Server ... 9-31
Maintenance Operations ... 9-31
Partition Maintenance Operations ... 9-32
Managing Indexes ... 9-38
Privileges for Partitioned Tables and Indexes ... 9-41
Auditing for Partitioned Tables and Indexes ... 9-42
SQL Extension: Partition-Extended Table Name ... 9-42
Examples of Partition-Extended Table Names ... 9-43

10 Built-In Datatypes

Oracle Datatypes ... 10-2
Character Datatypes ... 10-2
NUMBER Datatype ... 10-5
DATE Datatype ... 10-7
LOB Datatypes ... 10-9
RAW and LONG RAW Datatypes ... 10-11
ROWID Datatype ... 10-12
MLSLABEL Datatype ... 10-16
Summary of Oracle Datatype Information ... 10-17
ANSI, DB2, and SQL/DS Datatypes ... 10-19
Data Conversion ... 10-20

11 User-Defined Datatypes (Objects Option)

Introduction 11-2
Complex Data Models 11-2
Multimedia Datatypes 11-3
User-Defined Datatypes 11-3
Object Types 11-4
Collection Types 11-9
Application Interfaces 11-11
SQL 11-12
PL/SQL 11-12
Pro*C/C++ 11-12
OCI 11-13
OTT 11-14

12 Using User-Defined Datatypes

References and Name Resolution 12-2
Table Aliases 12-2
Method Calls without Arguments 12-3
Storage of User-Defined Types 12-4
Leaf-Level Attributes 12-4
Row Objects 12-4
Column Objects 12-5
REFs 12-5
Nested Tables 12-5
VARRAYs 12-5
Properties of Object Attributes 12-6
Nulls 12-6
Defaults 12-7
Constraints 12-8
Indexes 12-9
Triggers 12-9
Privileges on User-Defined Types and Their Methods 12-10
System Privileges 12-10
Schema Object Privileges 12-10
Using Types in New Types or Tables 12-11

Example 12-11
Privileges on Type Access and Object Access 12-12
Dependencies and Incomplete Types 12-13
Completing Incomplete Types 12-14
Type Dependencies of Tables 12-15
Import/Export of User-Defined Types 12-15

13 Object Views

Introduction 13-2
Advantages of Object Views 13-2
Defining Object Views 13-2
Using Object Views 13-4
Updating Object Views 13-4

Part V Data Access

14 SQL and PL/SQL

Structured Query Language (SQL) 14-2
SQL Statements 14-3
Identifying Nonstandard SQL 14-6
Recursive SQL 14-6
Cursors 14-6
Shared SQL 14-7
Parsing 14-7
SQL Processing 14-8
Overview of SQL Statement Execution 14-8
DML Statement Processing 14-10
DDL Statement Processing 14-14
Controlling Transactions 14-14
PL/SQL 14-15
How PL/SQL Executes 14-15
Language Constructs for PL/SQL 14-17
Stored Procedures 14-18
External Procedures 14-19

15 Transaction Management

Introduction to Transactions 15-2
Statement Execution and Transaction Control 15-3
Statement-Level Rollback 15-4
Oracle and Transaction Management 15-4
Committing Transactions 15-5
Rolling Back Transactions 15-6
Savepoints 15-7
The Two-Phase Commit Mechanism 15-7
Discrete Transaction Management 15-8

16 Advanced Queuing

Introduction to Message Queuing 16-3
Synchronous Communication 16-3
Asynchronous Communication 16-3
Oracle Advanced Queuing 16-4
Queuing Entities 16-4
Features of Advanced Queuing 16-6

17 Procedures and Packages

An Introduction to Stored Procedures and Packages 17-2
Stored Procedures and Functions 17-2
Packages 17-4
Procedures and Functions 17-6
Procedure Guidelines 17-7
Benefits of Procedures 17-7
Anonymous PL/SQL Blocks vs. Stored Procedures 17-8
Standalone Procedures 17-9
Dependency Tracking for Stored Procedures 17-9
External Procedures 17-9
Packages 17-10
Benefits of Packages 17-13
Dependency Tracking for Packages 17-14

How Oracle Stores Procedures and Packages 17-15
Compiling Procedures and Packages 17-15
Storing the Compiled Code in Memory 17-15
Storing Procedures or Packages in Database 17-15
How Oracle Executes Procedures and Packages 17-16
Verifying User Access 17-16
Verifying Procedure Validity 17-16
Executing a Procedure 17-17

18 Database Triggers

An Introduction to Triggers 18-2
How Triggers Are Used 18-3
Some Cautionary Notes about Triggers 18-3
Triggers vs. Declarative Integrity Constraints 18-5
Parts of a Trigger 18-5
Triggering Event or Statement 18-6
Trigger Restriction 18-7
Trigger Action 18-7
Types of Triggers 18-7
Row Triggers and Statement Triggers 18-7
BEFORE and AFTER Triggers 18-8
Trigger Combinations 18-9
INSTEAD OF Triggers 18-11
Trigger Execution 18-14
The Execution Model for Triggers and Integrity Constraint Checking 18-14
Data Access for Triggers 18-16
Storage of Triggers 18-17
Execution of Triggers 18-17
Dependency Maintenance for Triggers 18-18

19 Oracle Dependency Management

An Introduction to Dependency Issues 19-2
Resolving Schema Object Dependencies 19-4
Compiling Views and PL/SQL Program Units 19-5
Dependency Management and Nonexistent Schema Objects 19-7

Shared SQL Dependency Management 19-8
Local and Remote Dependency Management 19-8
Managing Local Dependencies 19-9
Managing Remote Dependencies 19-9

20 The Optimizer

What Is Optimization? 20-2
Execution Plans 20-2
Execution Order 20-5
Cost-Based and Rule-Based Optimization 20-6
The Cost-Based Approach 20-6
Overview of Optimizer Operations 20-12
Optimizer Operations 20-12
Types of SQL Statements 20-13
Evaluation of Expressions and Conditions 20-14
Constants 20-14
LIKE Operator 20-15
IN Operator 20-15
ANY or SOME Operator 20-15
ALL Operator 20-16
BETWEEN Operator 20-17
NOT Operator 20-17
Transitivity 20-17
Transforming and Optimizing Statements 20-19
Transforming ORs into Compound Queries 20-19
Transforming Complex Statements into Join Statements 20-22
Optimizing Statements That Access Views 20-24
Optimizing Compound Queries 20-36
Optimizing Distributed Statements 20-39
Choosing an Optimization Approach and Goal 20-40
The OPTIMIZER_MODE Initialization Parameter 20-40
Statistics in the Data Dictionary 20-41
The OPTIMIZER_GOAL Parameter of the ALTER SESSION Command 20-41
The FIRST_ROWS, ALL_ROWS, CHOOSE, and RULE Hints 20-42
PL/SQL and the Optimizer Goal 20-42

Choosing Access Paths 20-42
Access Methods 20-43
Access Paths 20-45
Choosing Among Access Paths 20-58
Optimizing Join Statements 20-63
Join Operations 20-63
Choosing Execution Plans for Join Statements 20-69
Views in Outer Joins 20-72
Optimizing Anti-Joins and Semi-Joins 20-74
Optimizing "Star" Queries 20-75
Star Query Example 20-75
Tuning Star Queries 20-75
Star Transformation 20-76

Part VI Parallel SQL and Direct-Load INSERT

21 Direct-Load INSERT

Introduction to Direct-Load INSERT 21-2
Advantages of Direct-Load INSERT 21-2
INSERT ... SELECT Statements 21-3
Varieties of Direct-Load INSERT Statements 21-3
Serial and Parallel INSERT 21-3
Logging Mode 21-5
Additional Considerations for Direct-Load INSERT 21-8
Index Maintenance 21-8
Space Considerations 21-8
Locking Considerations 21-9
Restrictions on Direct-Load INSERT 21-9

22 Parallel Execution

Overview of Parallel Execution 22-2
Operations That Can Be Parallelized 22-2
How Oracle Parallelizes Operations 22-3

Process Architecture for Parallel Execution 22-5
The Parallel Server Pool 22-7
Parallelizing SQL Statements 22-9
Setting the Degree of Parallelism 22-13
Determining the Degree of Parallelism for Operations 22-13
Balancing the Work Load 22-16
Parallelization Rules for SQL Statements 22-17
Parallel DDL 22-25
DDL Statements That Can Be Parallelized 22-25
CREATE TABLE ... AS SELECT in Parallel 22-26
Recoverability and Parallel DDL 22-27
Space Management for Parallel DDL 22-27
Parallel DML 22-29
Advantages of Parallel DML over Manual Parallelism 22-30
When to Use Parallel DML 22-31
Enabling Parallel DML 22-32
Transaction Model for Parallel DML 22-33
Recovery for Parallel DML 22-34
Space Considerations for Parallel DML 22-35
Lock and Enqueue Resources for Parallel DML 22-36
Restrictions on Parallel DML 22-37
Affinity 22-40
Other Types of Parallelism 22-42

Part VII Data Protection

23 Data Concurrency and Consistency

Data Concurrency and Consistency in a Multiuser Environment 23-2
Preventable Phenomena and Transaction Isolation Levels 23-2
Locking Mechanisms 23-3
How Oracle Manages Data Concurrency and Consistency 23-4
Multiversion Concurrency Control 23-4
Statement-Level Read Consistency 23-5
Transaction-Level Read Consistency 23-6
Oracle Isolation Levels 23-6

Setting the Isolation Level 23-7
Comparing Read Committed and Serializable Isolation 23-9
Choosing an Isolation Level 23-12
How Oracle Locks Data 23-14
Transactions and Data Concurrency 23-15
Deadlocks 23-16
Types of Locks 23-18
DML (Data) Locks 23-19
DDL Locks (Dictionary Locks) 23-26
Latches and Internal Locks 23-28
Explicit (Manual) Data Locking 23-29
Oracle Lock Management Services 23-40

24 Data Integrity

Definition of Data Integrity 24-2
Types of Data Integrity 24-2
How Oracle Enforces Data Integrity 24-4
An Introduction to Integrity Constraints 24-5
Advantages of Integrity Constraints 24-5
The Performance Cost of Integrity Constraints 24-7
Types of Integrity Constraints 24-7
NOT NULL Integrity Constraints 24-7
UNIQUE Key Integrity Constraints 24-8
PRIMARY KEY Integrity Constraints 24-10
FOREIGN KEY (Referential) Integrity Constraints 24-12
CHECK Integrity Constraints 24-16
The Mechanisms of Constraint Checking 24-17
Default Column Values and Integrity Constraint Checking 24-19
Deferred Constraint Checking 24-19
Constraint Attributes 24-20
SET CONSTRAINTS Mode 24-20
Unique Constraints and Indexes 24-21
Enabled, Disabled, and Enable Novalidate Constraints 24-21

25 Controlling Database Access

Database Security 25-2
Schemas, Database Users, and Security Domains 25-2
User Authentication 25-3
Authentication by the Operating System 25-3
Authentication by the Network 25-4
Authentication by the Oracle Database 25-4
Database Administrator Authentication 25-6
User Tablespace Settings and Quotas 25-8
Default Tablespace 25-8
Temporary Tablespace 25-8
Tablespace Access and Quotas 25-8
The User Group PUBLIC 25-9
User Resource Limits and Profiles 25-10
Types of System Resources and Limits 25-10
Profiles 25-13
Licensing 25-14
Concurrent Usage Licensing 25-14
Named User Licensing 25-15

26 Privileges and Roles

Privileges 26-2
System Privileges 26-2
Schema Object Privileges 26-3
Roles 26-10
Common Uses for Roles 26-11
The Mechanisms of Roles 26-13
Granting and Revoking Roles 26-13
Who Can Grant or Revoke Roles? 26-13
Naming Roles 26-14
Security Domains of Roles and Users 26-14
Named PL/SQL Blocks and Roles 26-14
Data Definition Language Statements and Roles 26-14
Predefined Roles 26-16

The Operating System and Roles 26-16
Roles in a Distributed Environment 26-16

27 Auditing

Introduction to Auditing 27-2
Auditing Features 27-2
Auditing Mechanisms 27-4
Statement Auditing 27-7
Privilege Auditing 27-7
Schema Object Auditing 27-8
Schema Object Audit Options for Views and Procedures 27-8
Focusing Statement, Privilege, and Schema Object Auditing 27-9
Auditing Successful and Unsuccessful Statement Executions 27-9
Auditing BY SESSION versus BY ACCESS 27-10
Auditing By User 27-12

28 Database Recovery

An Introduction to Database Recovery 28-2
Errors and Failures 28-2
Structures Used for Database Recovery 28-7
Database Backups 28-7
The Redo Log 28-7
Rollback Segments 28-8
Control Files 28-8
Rolling Forward and Rolling Back 28-8
The Redo Log and Rolling Forward 28-9
Rollback Segments and Rolling Back 28-9
Recovery Manager 28-10
Recovery Catalog 28-10
Parallelization 28-12
Report Generation 28-12
Performing Recovery in Parallel 28-13
Situations That Benefit from Parallel Recovery 28-14
Recovery Processes 28-14

Database Archiving Modes ... 28-16
NOARCHIVELOG Mode (Media Recovery Disabled) ... 28-16
ARCHIVELOG Mode (Media Recovery Enabled) ... 28-16
Control Files ... 28-19
Control File Contents ... 28-19
Multiplexed Control Files ... 28-20
Database Backups ... 28-21
Whole Database Backups ... 28-21
Partial Database Backups ... 28-22
The Export and Import Utilities ... 28-23
Read-Only Tablespaces and Backup ... 28-23
Survivability ... 28-24
Planning for Disaster Recovery ... 28-24
Standby Database ... 28-24

Part VIII Distributed Processing and Distributed Databases

29 Distributed Processing

Oracle Client/Server Architecture ... 29-2
Distributed Processing ... 29-2
Net8 ... 29-5
How Net8 Works ... 29-5

30 Distributed Databases

Oracle's Distributed Database Architecture ... 30-2
Clients and Servers ... 30-2
The Network ... 30-4
Databases and Database Links ... 30-4
Database Links ... 30-6
Schema Object Name Resolution ... 30-6
Connecting Between Oracle Server Versions ... 30-7
Distributed Databases and Distributed Processing ... 30-7
Distributed Databases and Database Replication ... 30-7
Heterogeneous Distributed Databases ... 30-8

Transparent SQL Access 30-8
Procedural Access 30-8
Gateway Features 30-9
Version 8 Gateways 30-10
Version 4 Gateways 30-10
Developing Distributed Database Applications 30-10
Remote and Distributed SQL Statements 30-10
Remote Procedure Calls (RPCs) 30-11
Remote and Distributed Transactions 30-11
Transparency in a Distributed Database System 30-13
Administering an Oracle Distributed Database System 30-15
Site Autonomy 30-15
Distributed Database Security 30-16
Tools for Administering Oracle Distributed Databases 30-17
Oracle Enterprise Manager 30-18
Third-Party Administration Tools 30-18
SNMP Support 30-19
National Language Support 30-19

31 Database Replication

What Is Replication? 31-2
Basic Replication 31-2
Advanced (Symmetric) Replication 31-3
Basic Replication Concepts 31-4
Uses of Basic Replication 31-4
Read-Only Table Snapshots 31-6
Snapshot Refreshes 31-8
Other Basic Replication Options 31-10
Advanced Replication Concepts 31-11
Uses for Advanced Replication 31-12
Advanced Replication Configurations 31-13
Advanced Replication and the Oracle Replication Manager 31-17
Replication Objects, Groups, Sites, and Catalogs 31-17
Oracle's Advanced Replication Architecture 31-19
Replication Administrators, Propagators, and Receivers 31-22

Replication Conflicts 31-22
Unique Advanced Replication Options..... 31-26

Part IX Appendix

A Operating System-Specific Information

Index

Send Us Your Comments

Oracle8 Concepts, Release 8.0

Part No. A58227-01

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

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

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

- electronic mail - infodev@us.oracle.com
- FAX - (650) 506-7200 Attn: Oracle Server Documentation
- postal service:
  Oracle Corporation
  Oracle Server Documentation Manager
  500 Oracle Parkway
  Redwood Shores, CA 94065
  USA

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

Preface

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

Oracle8 and Oracle8 Enterprise Edition

Oracle8 Concepts contains information that describes the features and functionality of the Oracle8 and the Oracle8 Enterprise Edition products. Oracle8 and Oracle8 Enterprise Edition have the same basic features. However, several advanced features are available only with the Enterprise Edition, and some of these are optional. For example, to use object functionality, you must have the Enterprise Edition and the Objects Option.

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

Audience

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

What You Should Already Know

You should be familiar with relational database concepts and with the operating system environment under which you are running Oracle.

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

If You'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 *Oracle8 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 *Oracle8 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 *Oracle8 Application Developer's Guide* and to the documentation for the tool or language product they are using to develop Oracle database applications.

How This Manual Is Organized

This manual is divided into the following parts:

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

- Part V: Data Access
- Part VI: Parallel SQL and Direct-Load INSERT
- Part VII: Data Protection
- Part VIII: Distributed Processing and Distributed Databases
- Part IX: Appendix

Part I: What Is Oracle?

Chapter 1: Introduction to the Oracle Server

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

Part II: Database Structures

Chapter 2: Data Blocks, Extents, and Segments

This chapter discusses how data is stored and how storage space is allocated for and consumed by various objects within an Oracle database. The space management background information here supplements that in the following chapter and in Chapter 8, “Schema Objects”.

Chapter 3: Tablespaces and Datafiles

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

Chapter 4: The Data Dictionary

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

Part III: The Oracle Instance

Chapter 5: Database and Instance Startup and Shutdown

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

Chapter 6: Memory Structures

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

Chapter 7: Process Structure

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

Part IV: The Object-Relational DBMS

Chapter 8: Schema Objects

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

Chapter 9: Partitioned Tables and Indexes

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

Chapter 10: Built-In Datatypes

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

Chapter 11: User-Defined Datatypes (Objects Option)

This chapter gives an overview of the object extensions provided by the Oracle object-relational database management system (ORDBMS).

Chapter 12: Using User-Defined Datatypes

This chapter describes the user-defined object types that are available in the Oracle ORDBMS.

Chapter 13: Object Views

This chapter describes the extensions to views provided by the Oracle ORDBMS.

Part V: Data Access

Chapter 14: 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, the Oracle procedural language extension to SQL.

Chapter 15: 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 16: Advanced Queuing

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

Chapter 17: Procedures and Packages

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

Chapter 18: 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 19: Oracle Dependency Management

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

Chapter 20: 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: Parallel SQL and Direct-Load INSERT

Chapter 21: Direct-Load INSERT

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

Chapter 22: Parallel Execution

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

Part VII: Data Protection

Chapter 23: Data Concurrency and Consistency

This chapter explains how Oracle provides concurrent access to and maintains the accuracy of shared information in a multiuser environment. It describes the auto-

matic mechanisms that Oracle uses to guarantee that the concurrent operations of multiple users do not interfere with each other.

Chapter 24: Data Integrity

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

Chapter 25: Controlling Database Access

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

Chapter 26: Privileges and Roles

This chapter discusses security at the system and object levels.

Chapter 27: Auditing

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

Chapter 28: Database Recovery

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

Part VIII: Distributed Processing and Distributed Databases

Chapter 29: Distributed Processing

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

Chapter 30: Distributed Databases

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

Chapter 31: Database Replication

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

Part IX: Appendix

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 who are interested primarily in managing security should focus on the information presented in Part VII, “Data Protection”, particularly Chapter 25, “Controlling Database Access”, Chapter 26, “Privileges and Roles”, and Chapter 27, “Auditing”.

Conventions Used in This Manual

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

Text of the Manual

The text of this manual uses the following conventions.

UPPERCASE Characters

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

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

Italicized Characters

Italicized words within text are book titles or emphasized words.

Code Examples

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

For example:

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

Example statements may include punctuation, such as commas or quotation marks. All punctuation in example statements is required. All example statements termi-

nate with a semicolon (;). Depending on the application, a semicolon or other terminator may or may not be required to end a statement.

UPPERCASE in Code Examples

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

`lowercase` in Code Examples

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

Your Comments Are Welcome

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

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

infodev@us.oracle.com

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

Server Technologies Documentation Manager

Oracle Corporation
500 Oracle Parkway
Redwood Shores, CA 94065

Fax: (650) 506-7200

Part I

What Is Oracle?

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

- Chapter 1, “Introduction to the Oracle Server”

The rest of this manual describes the concepts that are summarized in Chapter 1 more thoroughly.

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

- Databases and Information Management
- Database Structure and Space Management
- Memory Structure and Processes
- Data Concurrency and Consistency
- Distributed Processing and Distributed Databases
- Startup and Shutdown Operations
- Database Security
- Database Backup and Recovery
- The Object-Relational Model for Database Management
- Data Access

> **Attention:** This chapter contains information relating to both Oracle8 and the Oracle8 Enterprise Edition. Some of the features and options documented in this chapter are available only if you have purchased the Oracle8 Enterprise Edition. See *Getting to Know Oracle8 and the Oracle8 Enterprise Edition* for information about the differences between Oracle8 and the Oracle8 Enterprise Edition.

Databases and Information Management

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

The Oracle server provides efficient and effective solutions with the following features:

client / server (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 8.0 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. 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 section describes a different part of the overall architecture.

The Oracle Server

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

instance. 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 data. 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.

For more information about database structures, see "Database Structure and Space Management" on page 1-8.

An Oracle Instance

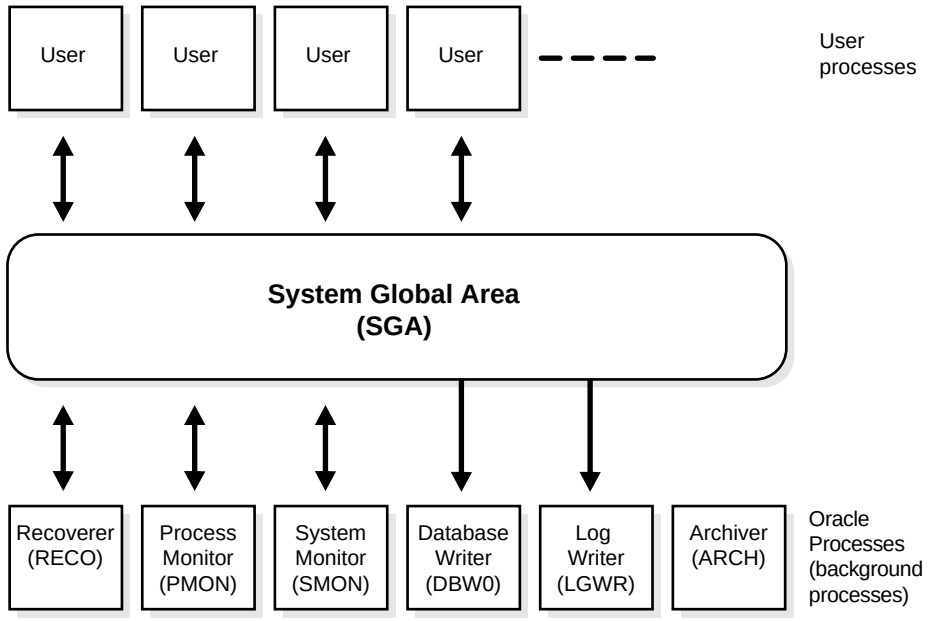
Every time a database is started, a system global area (SGA) is allocated and Oracle background processes are started. The system global area is a an area of memory used for database information shared by the database users. The combination of the background processes and memory buffers is called an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

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

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

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

Figure 1–1 An Oracle Instance



Communications Software and Net8

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 Net8. *Dispatchers* are optional background processes, present only when a multi-threaded server configuration is used. *Net8* is Oracle's interface to standard communications protocols that allows for the proper transmission of data between computers. See "Oracle and Net8" on page 1-26.

The Oracle Parallel Server: Multiple Instance Systems

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

Some hardware architectures (for example, shared disk systems) allow multiple computers to share access to data, software, or peripheral devices. Oracle with the Parallel Server option can take advantage of such architecture by running multiple instances that "share" a single physical database. In appropriate applications, the Oracle Parallel Server allows access to a single database by the users on multiple machines with increased performance.

> **Additional Information:** See *Oracle8 Parallel Server Concepts and Administration* for more information on the Oracle Parallel Server.

Oracle Databases

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*. See "Database Structure and Space Management" below for an overview of the logical and physical structures of the Oracle database.

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.

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.

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*.

Logical Database Structures

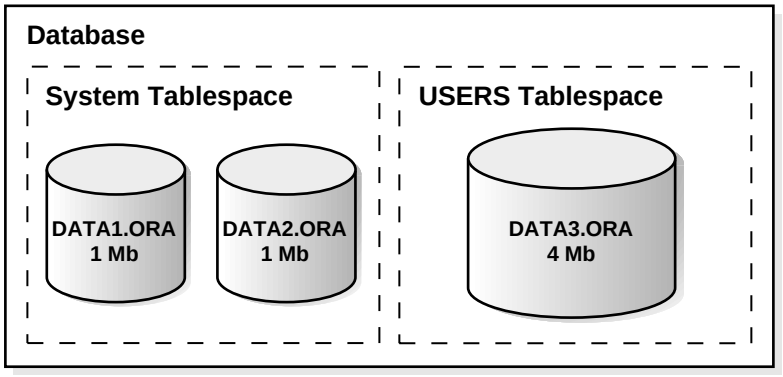
The following sections explain logical database structures, including tablespaces, schema objects, data blocks, extents, and segments.

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 some administrative operations.

Databases, Tablespaces, and Datafiles The relationship among databases, tablespaces, and datafiles (datafiles are described in the next section) is illustrated in Figure 1–2.

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 database objects. *Schema objects* are the logical structures that directly refer to the database's data. Schema objects include such structures as tables, views, sequences, stored procedures, synonyms, indexes, clusters, and database links. (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.) For more information about schema objects, see "Schemas and Schema Objects" on page 1-42.

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 2, "Data Blocks, Extents, and Segments".

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 1-21) ■ during database recovery (see "Database Backup and Recovery" on page 1-34) ■ 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 characteristics of datafiles are:

- 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 data-

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

Modified or new data is not necessarily written to a datafile immediately. To reduce the amount of disk access and increase performance, data is pooled in memory and written to the appropriate datafiles all at once, as determined by the DBW*n* background process of Oracle. (For more information about Oracle's memory and process structures and the algorithm for writing database data to the datafiles, see "Memory Structure and Processes" on page 1-13.)

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 1-34.

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 1-34 and Chapter 28, "Database Recovery" for more information.

Memory Structure and Processes

This section discusses the memory and process structures used by an Oracle server to manage a database. Among other things, the architectural features discussed in this section provide an understanding of the capabilities of the Oracle server to support:

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

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

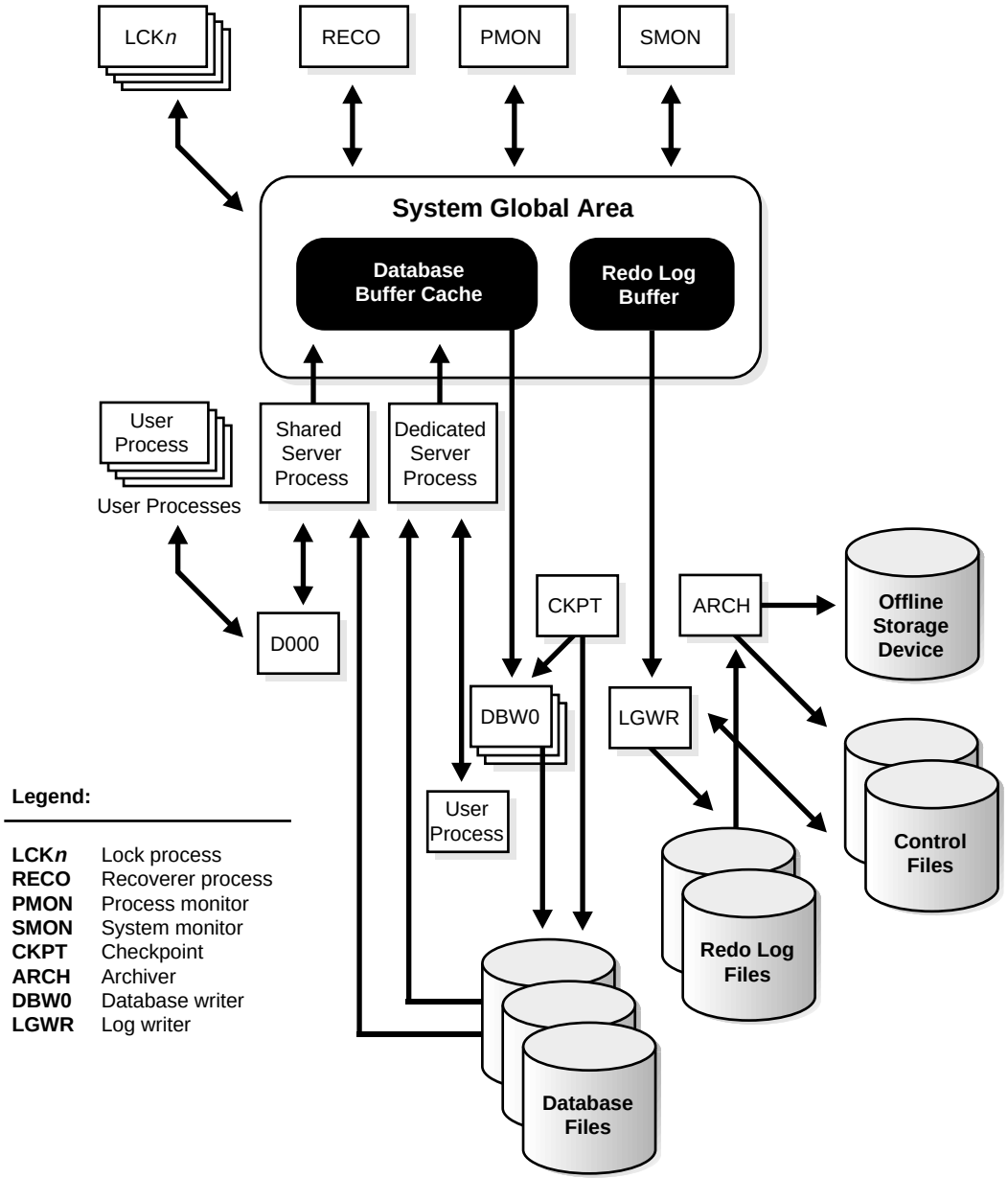
Processes are jobs or tasks that work in the memory of these computers.

Figure 1–3 "Memory Structures and Processes of Oracle" on page 1-14 shows a typical variation of the Oracle server memory and process structures.

Memory Structures

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

Figure 1–3 Memory Structures and Processes of Oracle



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 "An Oracle Instance" on page 1-6 and "Background Processes" on page 1-17 for more information.)

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*. The buffer cache contains modified as well as unmodified blocks. Because the most recently (and often the most frequently) used data is kept in memory, less disk I/O is necessary and performance is improved.

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

Shared Pool The shared pool is a portion of the system global area that contains shared memory constructs such as shared SQL areas. A shared SQL area is required to process every unique SQL statement submitted to a database (see "SQL Statements" on page 1-48). 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.

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

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

Program Global Area (PGA)

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

Process Architecture

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

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

User (Client) Processes

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

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

Oracle Process Architecture

Oracle processes are called by other processes to perform functions on behalf of the invoking process. The different types of Oracle processes and their specific functions are discussed in the following sections. They include server processes and background processes.

Server Processes

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

bal 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 *multithreaded 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 multithreaded server or if the user and server processes run on different machines, the user and server processes must be separate. Client/server systems separate the user and server processes and execute them on different machines.

Background Processes

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

An SGA and the set of Oracle background processes constitute an Oracle instance. (For information about the SGA, see "An Oracle Instance" on page 1-6 and "System Global Area (SGA)" on page 1-15.) Each Oracle instance may use several background processes. The names of these processes are DBW*n*, LGWR, CKPT, SMON, PMON, ARCH, RECO, D*nnn*, LCK*n*, SNP*n*, and QMN*n*.

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

Since Oracle uses write-ahead logging, DBW*n* does not need to write blocks when a transaction commits (see "Transactions" on page 1-49). Instead, DBW*n* is designed to perform batched writes with high efficiency. In the most common case, DBW*n* writes only when more data needs to be read into the system global area and too few database buffers are free. The least recently used data is written to the datafiles first. DBW*n* also performs writes for other functions such as checkpointing.

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 DBW*n*; this event is called a checkpoint. The *Checkpoint* process is responsible for signalling DBW*n* at checkpoints and updating all the datafiles and control files of the database to indicate the most recent checkpoint.

System Monitor (SMON) The *system monitor* performs instance recovery at instance startup. In a multiple instance system (one that uses Oracle 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 1-37.)

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*) The *lock* processes (LCK0, . . ., LCK9) are used for inter-instance locking in the Oracle Parallel Server; see "The Oracle Parallel Server: Multiple Instance Systems" on page 1-7 for information about this configuration.

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

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

The Program Interface

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

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

An Example of How Oracle Works

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

1. An instance is currently running on the computer that is executing Oracle (often called the *host* or *database server*).
2. A computer running an application (a *local machine* or *client workstation*) runs the application in a user process. The client application attempts to establish a connection to the server using the proper Net8 driver.
3. The server is running the proper Net8 driver. The server detects the connection request from the application and creates a (dedicated) server process on behalf of the user process.

4. The user executes a SQL statement and commits the transaction. For example, the user changes a name in a row of a table.
5. The server process receives the statement and checks the shared pool for any shared SQL area that contains an identical SQL statement. If a shared SQL area is found, the server process checks the user's access privileges to the requested data and the previously existing shared SQL area is used to process the statement; if not, a new shared SQL area is allocated for the statement so that it can be parsed and processed.
6. The server process retrieves any necessary data values from the actual datafile (table) or those stored in the system global area.
7. The server process modifies data in the system global area. The DBW*n* process writes modified blocks permanently to disk when doing so is efficient. Because the transaction committed, the LGWR process immediately records the transaction in the online redo log file.
8. If the transaction is successful, the server process sends a message across the network to the application. If it is not successful, an appropriate error message is transmitted.
9. Throughout this entire procedure, the other background processes run, watching for conditions that require intervention. In addition, the database server manages other users' transactions and prevents contention between transactions that request the same data.

These steps describe only the most basic level of operations that Oracle performs. (See Chapter 7, "Process Structure".)

Data Concurrency and Consistency

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

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

Concurrency

A primary concern of a multiuser database management system is how to control *concurrency*, or the simultaneous access of the same data by many users. Without

adequate concurrency controls, data could be updated or changed improperly, compromising data integrity.

If many people are accessing the same data, one way of managing data concurrency is to make each user wait his or her turn. The goal of a database management system is to reduce that wait so it is either non-existent or negligible to each user. All data manipulation (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 1-51, 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 that start *after* the user's transaction is committed only see the changes made by the committed transaction.

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

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

Read-Only Transactions

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

If you want to run a number of queries against multiple tables and if you are doing no updating, you may prefer a *read-only transaction*. After indicating that your transaction is read-only, you can execute as many queries as you like against any table, knowing that the results of each query are consistent with respect to the same point in time.

Locking Mechanisms

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

Locks are used to achieve two important database goals:

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

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

Automatic Locking

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

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

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

Manual Locking

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

Distributed Processing and Distributed Databases

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

Client/Server Architecture: Distributed Processing

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

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

The Client

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

The Server

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

Distributed Databases

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

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

While a distributed database allows increased access to a large amount of data across a network, it must also provide the ability to hide the location of the data and the complexity of accessing it across the network. The distributed DBMS must also preserve the advantages of administrating each local database as though it were 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 nondistributed systems, transactions should be carefully planned to include a logical set of SQL statements that should all succeed or fail as a unit. Oracle's two-phase commit mechanism guarantees that no matter what type of system or network failure might occur, a distributed transaction either commits on all involved nodes or rolls back on all involved nodes to maintain data consistency across the global distributed database.

Complete Transparency to Database Users The Oracle two-phase commit mechanism is completely transparent to users that issue distributed transactions. A simple COMMIT statement denoting the end of a transaction automatically triggers the two-phase commit mechanism to commit the transaction; no coding or complex statement syntax is required to include distributed transactions within the body of a database application.

Automatic Recovery from System or Network Failures The RECO (recoverer) background process automatically resolves the outcome of *in-doubt distributed transactions* — dis-

tributed transactions in which the commit was interrupted by any type of system or network failure. After the failure is repaired and communication is reestablished, the RECO of each local Oracle server automatically commits or rolls back any in-doubt distributed transactions consistently on all involved nodes.

Optional Manual Override Capability In the event of a long-term failure, Oracle allows each local administrator to manually commit or roll back any distributed transactions that are in doubt as a result of the failure. This option allows the local database administrator to free up any locked resources that may be held indefinitely as a result of the long-term failure.

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

Table Replication

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

Data can be replicated using snapshots or replicated master tables. For more information, see Chapter 31, "Database Replication".

> **Additional Information:** *Oracle8 Replication* contains detailed information about database replication.

Oracle and Net8

Net8 is Oracle's mechanism for interfacing with the communication protocols used by the networks that facilitate distributed processing and distributed databases. Communication protocols define the way that data is transmitted and received on a network. In a networked environment, an Oracle database server communicates with client workstations and other Oracle database servers using Oracle software called Net8.

Net8 supports communications on all major network protocols, ranging from those supported by PC LANs to those used by the largest of mainframe computer systems.

Using Net8, the application developer does not have to be concerned with supporting network communications in a database application. If a new protocol is used, the database administrator makes some minor changes, while the application requires no modifications and continues to function.

Startup and Shutdown Operations

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

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

These steps are described in "Instance and Database Startup" on page 5-5.

When the Oracle server starts up, it uses a parameter file that contains initialization parameters. These parameters specify the name of the database, the amount of memory to allocate, the names of control files, and various limits and other system parameters. See "Parameter Files" on page 5-4 for a sample parameter file.

> **Additional Information:** Refer to *Oracle8 Reference* for more information about initialization parameters.

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

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

Oracle automatically performs all three steps when an instance is shut down. See "Database and Instance Shutdown" on page 5-8 for more information.

Database Security

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

- prevent unauthorized database access

- prevent unauthorized access to schema objects
- control disk usage
- control system resource 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 database 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 a user's schema objects
- the resource limits for a user

System security mechanisms check:

- whether a user is authorized to connect to the database
- whether database auditing is active
- which system operations a user can perform

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

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

Security Mechanisms

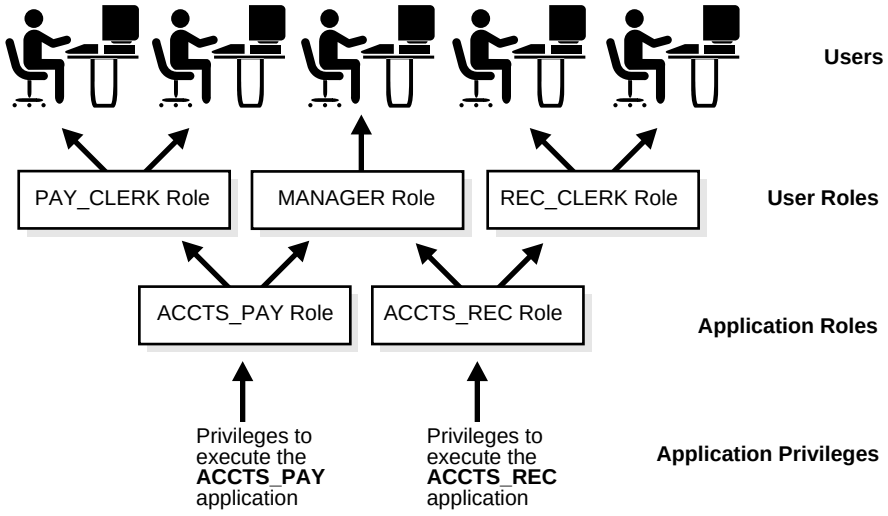
The Oracle server provides *discretionary access control*, which is a means of restricting access to information based on privileges. The appropriate privilege must be assigned to a user in order for that user to access a schema object. Appropriately privileged users can grant other users privileges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

Oracle manages database security using several different facilities:

- database users and schemas
- privileges
- roles
- storage settings and quotas
- resource limits
- auditing

Figure 1–4 illustrates the relationships of the different Oracle security facilities, and the following sections provide an overview of users, privileges, and roles.

Figure 1–4 Oracle Security Features



Database Users and Schemas

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

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

- actions (privileges and roles) available to the user
- 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 schema object privileges.

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

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

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

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

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

Roles

Oracle provides for easy and controlled privilege management through roles. *Roles* are named groups of related privileges that are granted to users or other roles. The following properties of roles allow for easier privilege management:

- *reduced granting of privileges* — Rather than explicitly granting the same set of privileges to many users, a database administrator can grant the privileges for a group of related users 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 schema object, the user's default tablespace is used if the user has the privilege to create the schema object and a quota in the specified default tablespace. The default tablespace feature provides Oracle with information to direct space usage in situations where schema object's location is not specified.

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

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

Profiles and Resource Limits

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

- number of concurrent sessions the user can establish
- CPU processing time
  - available to the user's session
  - 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
- password restrictions
  - account locking after multiple unsuccessful login attemts
  - password expiration and grace period
  - password reuse and complexity restrictions

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

Auditing

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

statement auditing	Statement auditing is the auditing of specific SQL statements without regard to specifically named schema objects. (In addition, database triggers allow a DBA to extend and customize Oracle's built-in auditing features.) Statement auditing can be broad and audit all users of the system or can be focused to audit only selected users of the system. For example, statement auditing by user can audit connections to and disconnections from the database by the users SCOTT and LORI.
privilege auditing	Privilege auditing is the auditing of the use of powerful system privileges without regard to specifically named schema objects. Privilege auditing can be broad and audit all users or can be focused to audit only selected users.
schema object auditing	Schema object auditing is the auditing of accesses to specific schema objects without regard to user. Object auditing monitors the statements permitted by object privileges, such as SELECT or DELETE statements on a given table.

For all types of auditing, Oracle allows the selective auditing of successful statement executions, unsuccessful statement executions, or both. This allows monitoring of suspicious statements, regardless of whether the user issuing a statement has the appropriate privileges to issue the statement.

The results of audited operations are recorded in a table referred to as the *audit trail*. Predefined views of the audit trail are available so that you can easily retrieve audit records.

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* (*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. A schema object's label indicates the sensitivity of the information that it contains. A user's label must meet certain criteria, determined by MAC policy, in order for him or her to be allowed to access a labeled schema object. Because this type of access control is always enforced above any discretionary controls implemented by users, this type of security is called "mandatory".

> **Additional Information:** See your *Trusted Oracle* documentation 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 1-12, the *redo log* is a set of files that protect altered database data in memory that has not been written to the datafiles. The redo log can consist of two parts: the online redo log and the archived redo log.

The Online Redo Log The *online redo log* is a set of two or more *online redo log files* that record all 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 "Data Blocks, Extents, and Segments" on page 1-10, 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.

Whole Database Backups A whole database 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 DBW*n* 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.

The Recovery Manager

The Recovery Manager is an Oracle utility that manages backup and recovery operations, creating backups of database files and restoring or recovering a database from backups.

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

The recovery catalog contains:

- information about backups of datafiles and archivelogs
- information about datafile copies
- information about archived redo logs and copies of them
- information about the physical schema of the target database
- named sequences of commands called *stored scripts*.

> **Additional Information:** See the *Oracle8 Backup and Recovery Guide* for more information about the Recovery Manager.

The Object-Relational Model for Database Management

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

The Relational Model

The relational model has three major aspects:

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

Relational database management systems offer benefits such as:

- independence of physical data storage and logical database structure
- variable and easy access to all data
- complete flexibility in database design
- reduced data storage and redundancy

The Object-Relational Model

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

Attention: User-defined object types are available only if you have purchased the Oracle8 Enterprise Edition with the Objects Option. See *Getting to Know Oracle8 and the Oracle8 Enterprise Edition* for details about the features and options available with Oracle8 Enterprise Edition.

Object types are abstractions of the real-world entities — for example, purchase orders — that application programs deal with. An object type has three kinds of components:

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

Schemas and Schema Objects

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

Tables

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

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

Oracle8 provides for the *partitioning* of tables. For more information, see Chapter 9, "Partitioned Tables and Indexes".

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 1-54.

Views

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

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

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

Views are often used to do the following:

- Provide an additional level of table security by restricting access to a predetermined set of rows and columns of a table. For example, a view of a table can be created so that columns with sensitive data (for example, salary information) are not included in the definition of the view.
- Hide data complexity. For example, a single view can combine 12 monthly sales tables to provide a year of data for analysis and reporting. A single view can also be used to create a *join*, which is a display of related columns or rows in multiple tables. However, the view hides the fact that this data actually originates from several tables.
- Simplify 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. See "Updatable Join Views" on page 8-13 for more information.

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, packages, triggers, and anonymous blocks.

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 1-48.

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 a schema object itself, but instead is a direct reference to a schema object. Synonyms are used to

- mask the real name and owner of a schema object
- provide public access to a schema object
- provide location transparency for tables, views, or program units of a remote database
- simplify the SQL statements for database users

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

Indexes, 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.

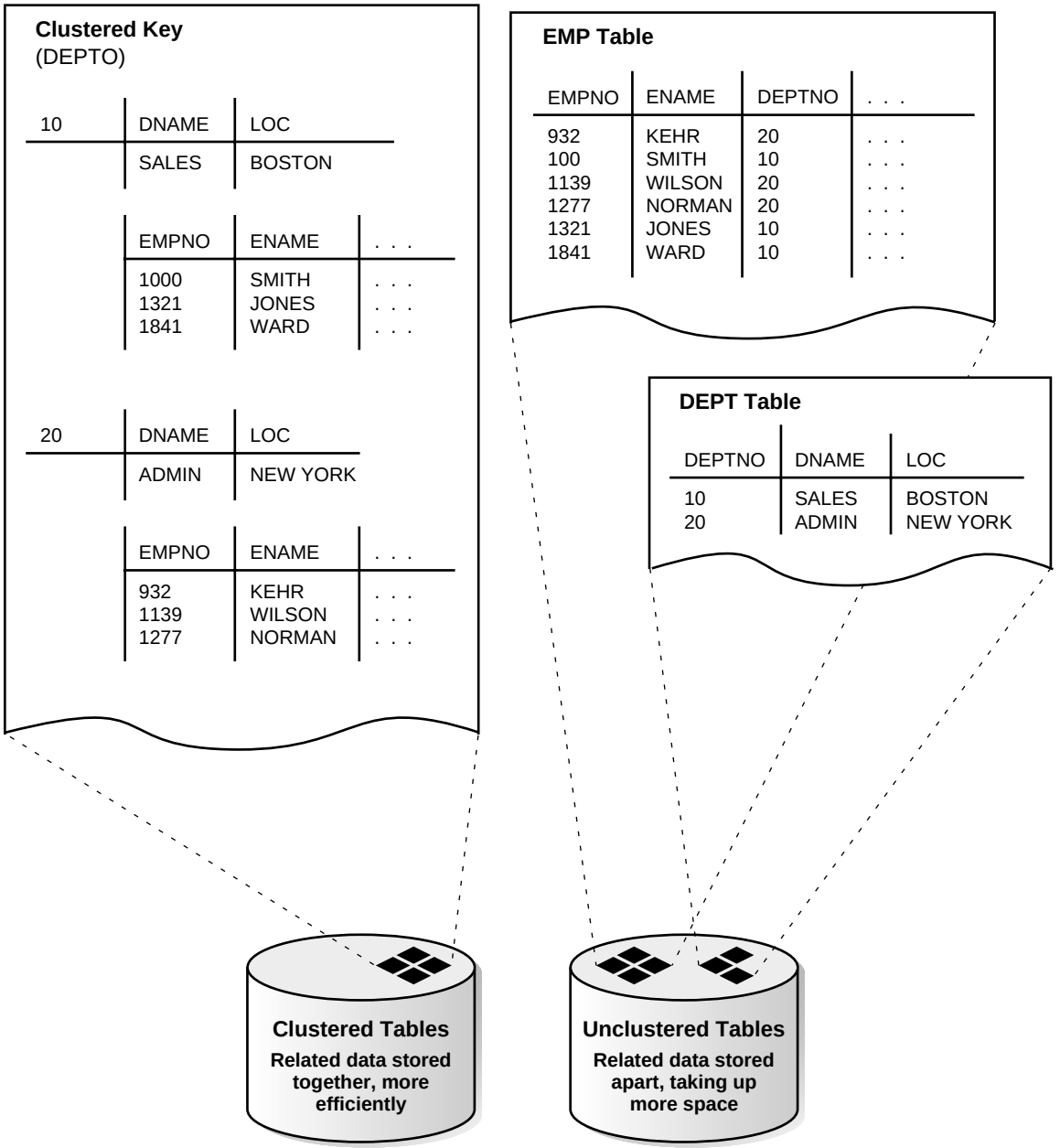
Oracle8 enables you to *partition* indexes. For more information, see Chapter 9, “Partitioned Tables and Indexes”.

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 1–5 illustrates how clustered and non-clustered data are physically stored.

Figure 1–5 Clustered and Unclustered Tables



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

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

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

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

Database Links

A *database link* is a named schema object that describes a "path" from one database to another. Database links are implicitly used when a reference is made to a *global object name* in a distributed database. See "Distributed Databases" on page 1-24 and Chapter 30, "Distributed Databases" for more information.

The Data Dictionary

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

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

A data dictionary is created when a database is created. To accurately reflect the status of the database at all times, the data dictionary is automatically updated by Oracle in response to specific actions (such as when the structure of the database is altered). The data dictionary is critical to the operation of the database, which relies

on the data dictionary to record, verify, and conduct ongoing work. For example, during database operation, Oracle reads the data dictionary to verify that schema objects exist and that users have proper access to them.

Data Access

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

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

SQL — The Structured Query Language

SQL is a simple, powerful database access language that is the standard language for relational database management systems. The SQL implemented by Oracle Corporation for Oracle is 100 percent compliant with the ANSI/ISO standard SQL data language.

SQL Statements

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

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

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

```
SELECT ename
```

A SQL statement can be thought of as a very simple, but powerful, computer program or instruction. SQL statements are divided into the following categories:

- Data Definition Language (DDL) statements
- Data Manipulation Language (DML) statements
- transaction control statements

- session control statements
- system control statements
- embedded SQL statements

Data Definition Statements (DDL) *DDL statements* define, maintain, and drop schema objects when they are no longer needed. DDL statements also include statements that permit a user to grant other users the *privileges*, or rights, to access the database and specific objects within the database. (See "Database Security" on page 1-27.)

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 1-49.) 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 state-

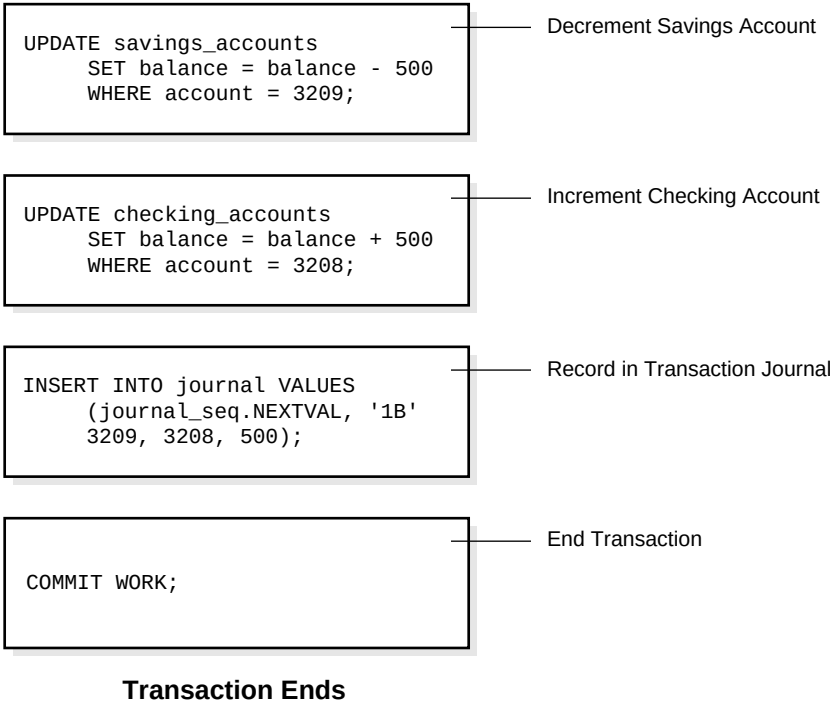
ment. A transaction ends when it is explicitly committed or rolled back (both terms are discussed later in this section) by that user.

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 1–6 illustrates the banking transaction example.

Figure 1–6 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, the users of PL/SQL can access data only as intended by the application developer (unless another access route is granted).
- PL/SQL blocks can be sent by an application to a database, executing complex operations without excessive network traffic.

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

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

Procedures and Functions

Procedures and *functions* consist of a set of SQL and PL/SQL statements that are grouped together as a unit to solve a specific problem or perform a set of related tasks. A procedure is created and stored in compiled form in the database and can be executed by a user or a database application. Procedures and functions are identical except that functions always return a single value to the caller, while procedures do not return values to the caller.

Packages

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

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.

Methods

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

> **Attention:** User-defined datatypes are available only if you have purchased the Oracle8 Enterprise Edition with the Objects Option. See *Getting to Know Oracle8 and the Oracle8 Enterprise Edition* for more information about Oracle8 Enterprise Edition.

Methods are different from stored procedures in two ways:

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

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

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

Data Integrity

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

Integrity Constraints

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

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

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

The following integrity constraints are supported by Oracle:

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

Keys

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.

Part II

Database Structures

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

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

Note: For more information about Oracle server structures, see:

- Chapter 6, “Memory Structures”
- Chapter 7, “Process Structure”
- Chapter 8, “Schema Objects”
- Chapter 9, “Partitioned Tables and Indexes”

2

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 among the logical storage structures in the Oracle server. It includes:

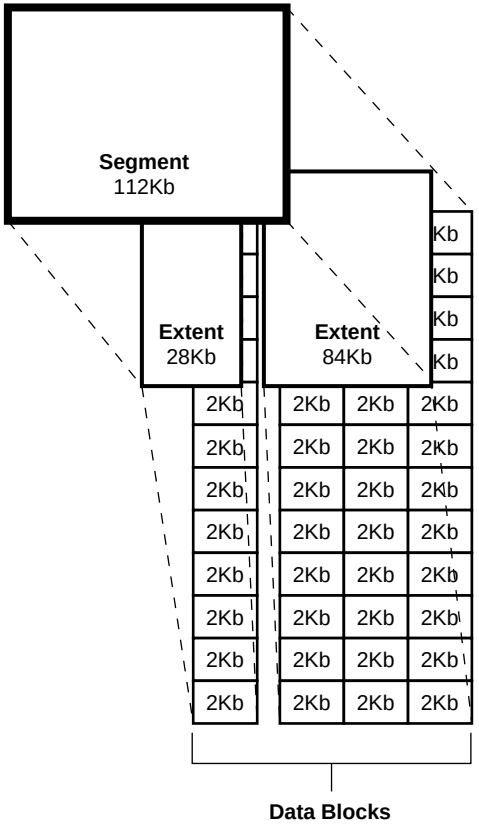
- The Relationships Among Data Blocks, Extents, and Segments
- Data Blocks
- Extents
- Segments

> **Additional Information:** If you are using Trusted Oracle, see your *Trusted Oracle* documentation for more information about storage in that environment.

The Relationships Among Data Blocks, Extents, and Segments

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

Figure 2–1 The Relationships Among Segments, Extents, and Data Blocks



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

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

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

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

A segment (and all its extents) ar stored in one tablespace. Within a tablespace, a segment can span datafiles (have extents with data from more than one file). However, each extent can contain data from only one datafile.

Data Blocks

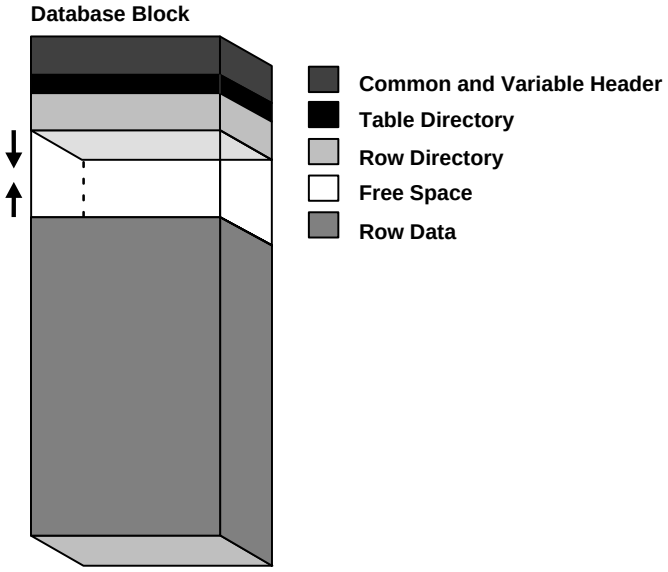
Oracle manages the storage space in the datafiles of a database in units called *data blocks*. A data block is the smallest unit of I/O used by a database. In contrast, at the physical, operating system level, all data is stored in bytes. Each operating system has what is called a *block size*. Oracle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

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

Data Block Format

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

Figure 2–2 Data Block Format



Header (Common and Variable)

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

Table Directory

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

Row Directory

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

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

Overhead

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

Row Data

This portion of the data block contains table or index data. Rows can span blocks; see "Row Chaining and Migrating" on page 2-10.

Free Space

Free space is allocated for insertion of new rows and for updates to rows that require additional space (for example, when a trailing null is updated to a non-null value). Whether issued insertions actually occur in a given data block is a function of current free space in that data block and the value of the space management parameter PCTFREE. The next section, "An Introduction to PCTFREE, PCTUSED, and Row Chaining", contains more information on space management parameters.

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

An Introduction to PCTFREE, PCTUSED, and Row Chaining

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

Note: This discussion does not apply to LOB datatypes (BLOB, CLOB, NCLOB, and BFILE) — they do not use the PCTFREE storage parameter or free lists. See "LOB Datatypes" on page 10-9 for more information.

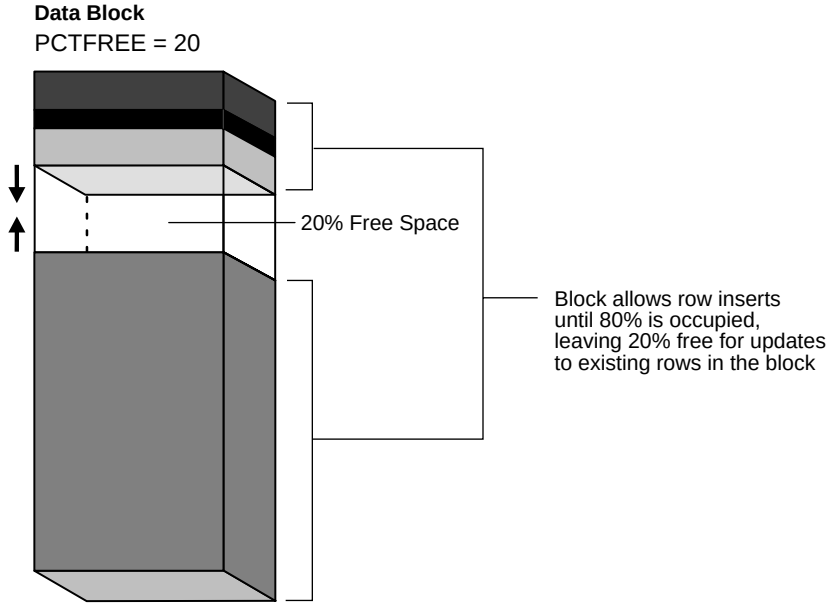
The PCTFREE Parameter

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

```
PCTFREE 20
```

This states that 20% of each data block in this table's data segment will be kept free and available for possible updates to the existing rows already within each block. New rows can be added to the row data area, and corresponding information can be added to the variable portions of the overhead area, until the row data and overhead total 80% of the total block size. Figure 2–3 illustrates PCTFREE.

Figure 2–3 PCTFREE



The PCTUSED Parameter

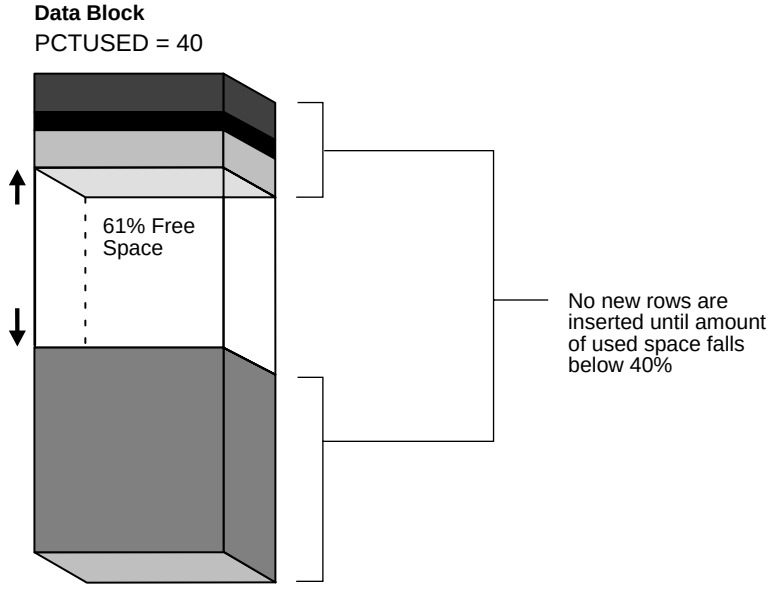
The PCTUSED parameter sets the minimum percentage of a block that can be *used* for row data plus overhead before new rows will be added to the block. After a

data block is filled to the limit determined by PCTFREE, Oracle considers the block unavailable for the insertion of new rows until the percentage of that block falls below the parameter PCTUSED. Until this value is achieved, Oracle uses the free space of the data block only for updates to rows already contained in the data block. For example, assume that you specify the following parameter in a CREATE TABLE statement:

```
PCTUSED 40
```

In this case, a data block used for this table's data segment is considered unavailable for the insertion of any new rows until the amount of used space in the block falls to 39% or less (assuming that the block's used space has previously reached PCTFREE). Figure 2–4 illustrates this.

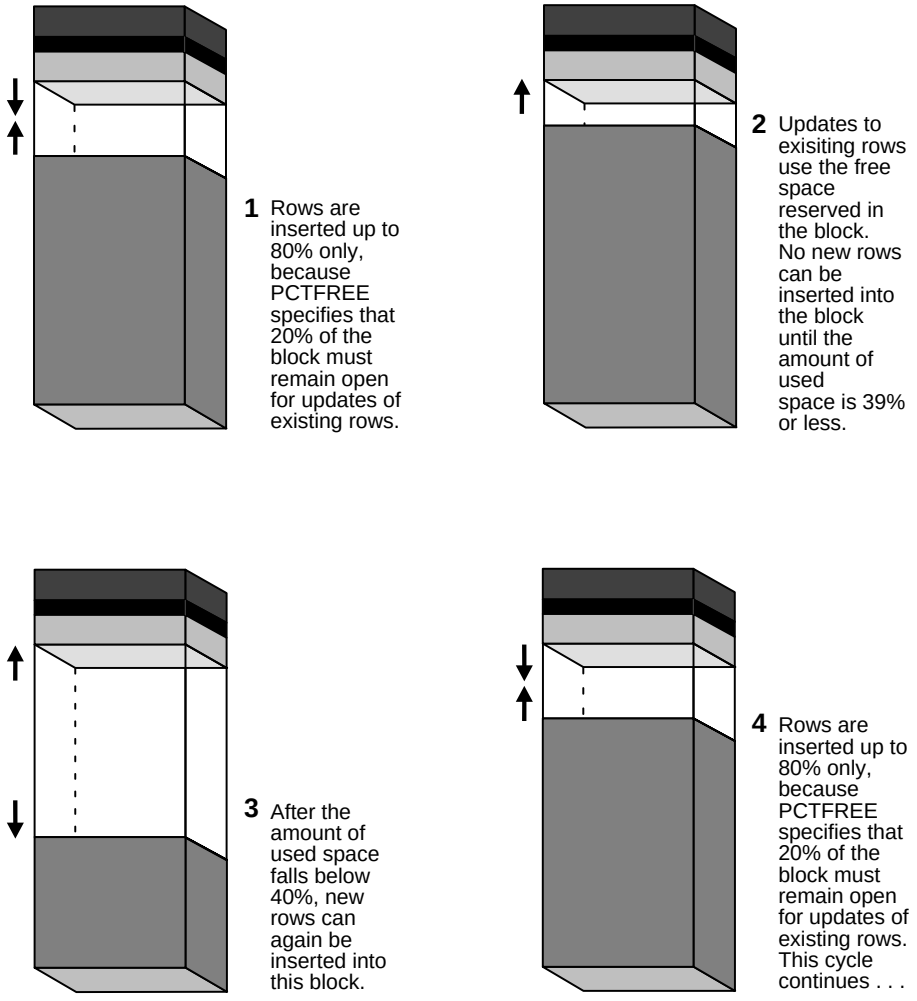
Figure 2–4 PCTUSED



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 2–5 illustrates the interaction of these two parameters.

Figure 2–5 Maintaining the Free Space of Data Blocks with PCTFREE and PCTUSED



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

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

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

Availability and Compression of Free Space in a Data Block

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

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

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

Row Chaining and Migrating

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

> **Note:** The format of a row and a row piece are described in "Row Format and Size" on page 8-5.

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

> **Note:** For more information about ROWID, see "ROWID Datatype" on page 10-12.

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.

> **Additional Information:** See *Oracle8 Tuning* for information about reducing chained and migrated rows and improving I/O performance.

Extents

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

When Extents Are Allocated

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

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

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

Rollback segments always have at least two extents. For more information, see "How Extents Are Used and Allocated for Rollback Segments" on page 2-19.

> **Note:** This chapter applies to serial operations, in which one server process parses and executes a SQL statement. Extents are allocated somewhat differently in parallel SQL statements, which entail multiple server processes. See "Free Space and Parallel DDL" on page 22-27 for more information.

Determining the Number and Size of Extents

Storage parameters expressed in terms of extents define every segment. Storage parameters apply to all types of segments. They control how Oracle allocates free database space for a given segment. For example, you can determine how much space is initially reserved for a table's data segment or you can limit the number of extents the table can allocate by specifying the storage parameters of a table in the STORAGE clause of the CREATE TABLE statement.

How Extents Are Allocated

Oracle controls the allocation of incremental extents for a given segment as follows:

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

   a. Oracle searches for a contiguous set of data blocks that matches the size of new extent plus one block to reduce internal fragmentation. (The size is

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

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

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

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

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

2. Once Oracle finds and allocates the necessary free space in the tablespace, it allocates a portion of the free space that corresponds to the size of the incremental extent. If Oracle found a larger amount of free space than was required for the extent, Oracle leaves the remainder as free space (no smaller than 5 contiguous blocks).

3. Oracle updates the segment header and data dictionary to show that a new extent has been allocated and that the allocated space is no longer free.

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

ALTER TABLE or ALTER CLUSTER statement, Oracle formats the extent's blocks when it allocates the extent.

When Extents Are Deallocated

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

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

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

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

> **Additional Information:** See *Oracle8 Administrator's Guide* and *Oracle8 SQL Reference* for more information on deallocating extents.

Nonclustered Tables

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

After you drop a nonclustered table, this space can be reclaimed when other extents require free space.

Oracle reclaims all the extents of 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

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

Snapshots and Snapshot Logs

Oracle deallocates the extents of snapshots and snapshot logs in the same manner as for nonclustered and clustered tables.

> **Additional Information:** See *Oracle8 Replication* for more information on snapshots and snapshot logs.

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), Oracle automatically deallocates one or more extents from the rollback segment. See "How Extents Are Deallocated from a Rollback Segment" on page 2-22 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. A single sort allocates its own temporary segment, in the temporary tablespace of the user issuing the statement, and then returns the extents to the tablespace.

Multiple sorts, however, can use sort segments in a temporary tablespace designated exclusively for sorts. These sort segments are allocated only once for the instance, and they are not returned after the sort but remain available for other multiple sorts. For more information, see "Temporary Segments" on page 2-16.

Segments

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

Oracle databases use four types of segments:

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

The following sections discuss each type of segment.

Data Segments

Every nonclustered table or partition and every cluster in an Oracle database has a single data segment to hold all of its data. Oracle creates this data segment when you create the nonclustered table or cluster with the CREATE command.

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

Note: Oracle creates segments for snapshots and snapshot logs in the same manner as for nonclustered and clustered tables.

Additional Information: See *Oracle8 Replication* for more information on snapshots and snapshot logs, and see *Oracle8 SQL Reference* for more information on the CREATE and ALTER commands.

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. In this command, you can specify storage parameters for the extents of the index segment and a tablespace in which to create the index segment. (The seg-

ments of a table and an index associated with it do not have to occupy the same tablespace.) Setting the storage parameters directly affects the efficiency of data retrieval and storage.

Temporary Segments

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

Operations Requiring Temporary Segments

The following 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

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

> **Additional Information:** See the *Oracle8 Reference* for information on SORT_AREA_SIZE and other initialization parameters.

How Temporary Segments Are Allocated

Oracle allocates temporary segments as needed during a user session, in the temporary tablespace of the user issuing the statement. You specify this tablespace with a CREATE USER or an ALTER USER command using the TEMPORARY TABLESPACE option. If no temporary tablespace has been defined for the user,

the default temporary tablespace is the SYSTEM tablespace. The default storage characteristics of the containing tablespace determine those of the extents of the temporary segment.

Oracle drops temporary segments when the statement completes.

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

For more information about assigning a user's temporary segment tablespace, see Chapter 25, "Controlling Database Access".

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

Rollback Segments

Each database contains one or more rollback segments. A rollback segment records the old values of data that was changed by each transaction (whether or not committed). Rollback segments are used to provide read consistency, to roll back transactions, and to recover the database. For specific information about how rollback segments function in these situations, see the appropriate sections of this book:

Topic	Section
Read Consistency	"Multiversion Concurrency Control" on page 23-4
Transaction Rollback	"Rolling Back Transactions" on page 15-6
Database Recovery	"Rollback Segments and Rolling Back" on page 28-9

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 filenumber and block ID corresponding to the data that was changed) and the data as it existed before an operation in a transaction. Oracle links rollback entries for the same transaction, so the entries can be found easily if necessary for transaction rollback.

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

Logging Rollback Entries

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

When Rollback Information Is Required

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

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

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

Transactions and Rollback Segments

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

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

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

When you commit a transaction, Oracle releases the rollback information but does not immediately destroy it. The information remains in the rollback segment to create read-consistent views of pertinent data for queries that started before the transaction committed. To guarantee that rollback data is available for as long as possible for such views, Oracle writes the extents of rollback segments sequentially. When the last extent of the rollback segment becomes full, Oracle continues writing rollback data by wrapping around to the first extent in the segment. A long-running transaction (idle or active) may require a new extent to be allocated for the rollback segment. See Figure 2–6, Figure 2–7, and Figure 2–8 for more information about how transactions use the extents of a rollback segment.

Each rollback segment can handle a fixed number of transactions from one instance. Unless you explicitly assign transactions to particular rollback segments, Oracle distributes active transactions across available rollback segments so that all rollback segments are assigned approximately the same number of active transactions. Distribution does **not** depend on the size of the available rollback segments. Therefore, in environments where all transactions generate the same amount of rollback information, all rollback segments can be the same size.

> **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.

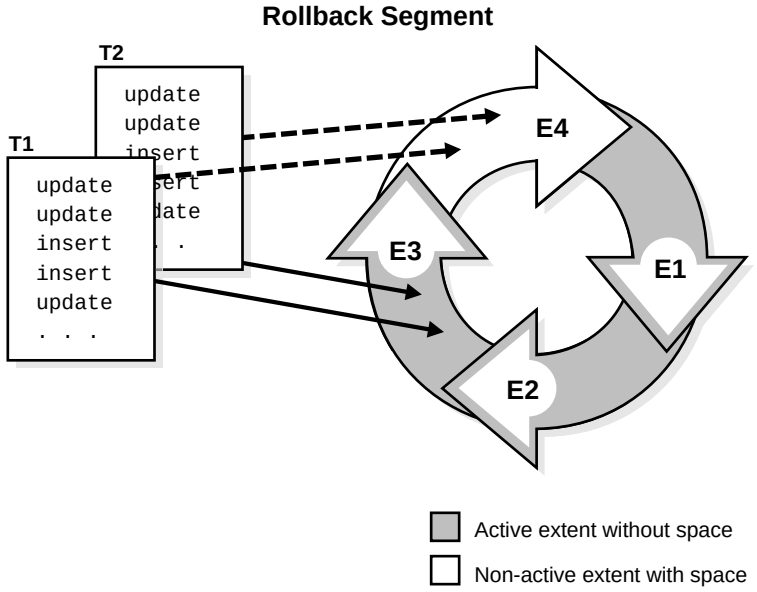
One transaction writes sequentially to a single rollback segment. Each transaction writes to only one extent of the rollback segment at any given time. Many *active* transactions can write concurrently to a single rollback segment—even the same extent of a rollback segment; however, each data block in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

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

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

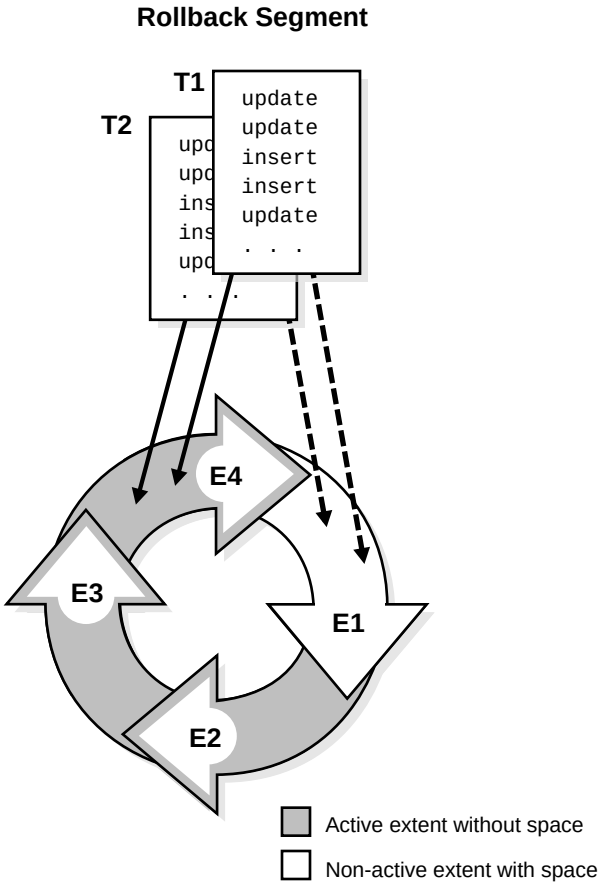
The first transaction that needs to acquire more rollback space checks the next extent of the rollback segment. If the next extent of the rollback segment does not contain information from an active transaction, Oracle makes it the current extent, and all transactions that need more space from then on can write rollback information to the new current extent. Figure 2–6 illustrates two transactions, T1 and T2, which begin writing in the third extent (E3) and continue writing to the fourth extent (E4) of a rollback segment.

Figure 2–6 Use of Allocated Extents in a Rollback Segment



As the transactions continue writing and fill the current extent, Oracle checks the next extent already allocated for the rollback segment to determine if it is available. In Figure 2–7, when E4 is completely full, T1 and T2 continue any further writing to the next extent allocated for the rollback segment that is available; in this figure, E1 is the next extent. This figure shows the cyclical nature of extent use in rollback segments.

Figure 2–7 Cyclical Use of the Allocated Extents in a Rollback Segment

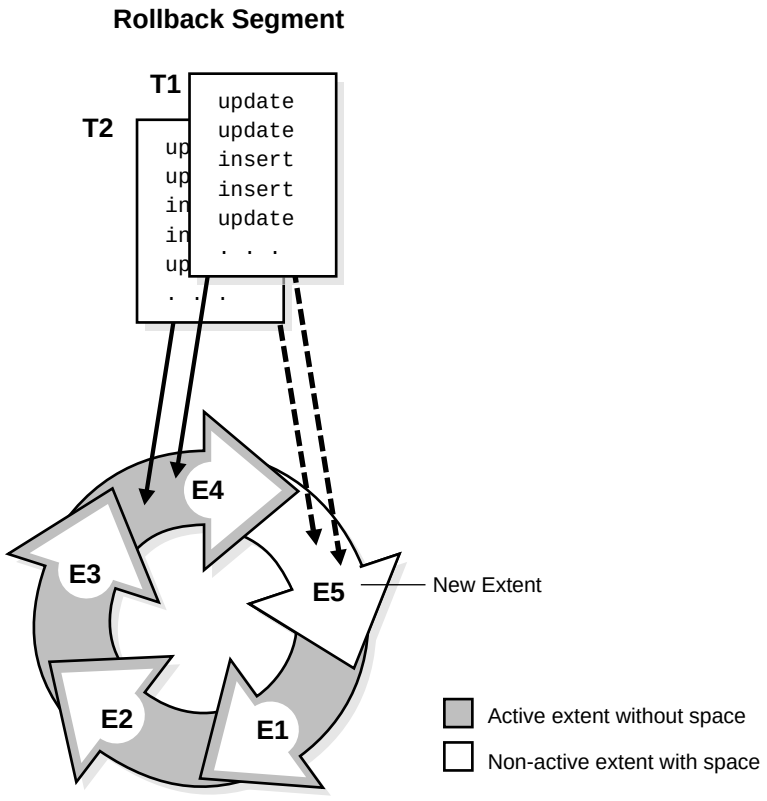


To continue writing rollback information for a transaction, Oracle always tries to reuse the next extent in the ring first. However, if the next extent contains data from active transaction, then Oracle must allocate a new extent. Oracle can allocate new extents for a rollback segment until the number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

Figure 2–8 shows a new extent allocated for a rollback segment. The uncommitted transactions are long running (either idle, active, or persistent in-doubt distributed transactions). At this time, they are writing to the fourth extent, E4, in the rollback

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

Figure 2–8 Allocation of a New Extent for a Rollback Segment



How Extents Are Deallocated from a Rollback Segment

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

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

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

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

The Rollback Segment SYSTEM

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

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

Oracle Instances and Types of Rollback Segments

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

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

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

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

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

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

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

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

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

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

> **Additional Information:** See *Oracle8 Parallel Server Concepts and Administration* for more information about rollback segment use in an Oracle Parallel Server.

Rollback Segment States

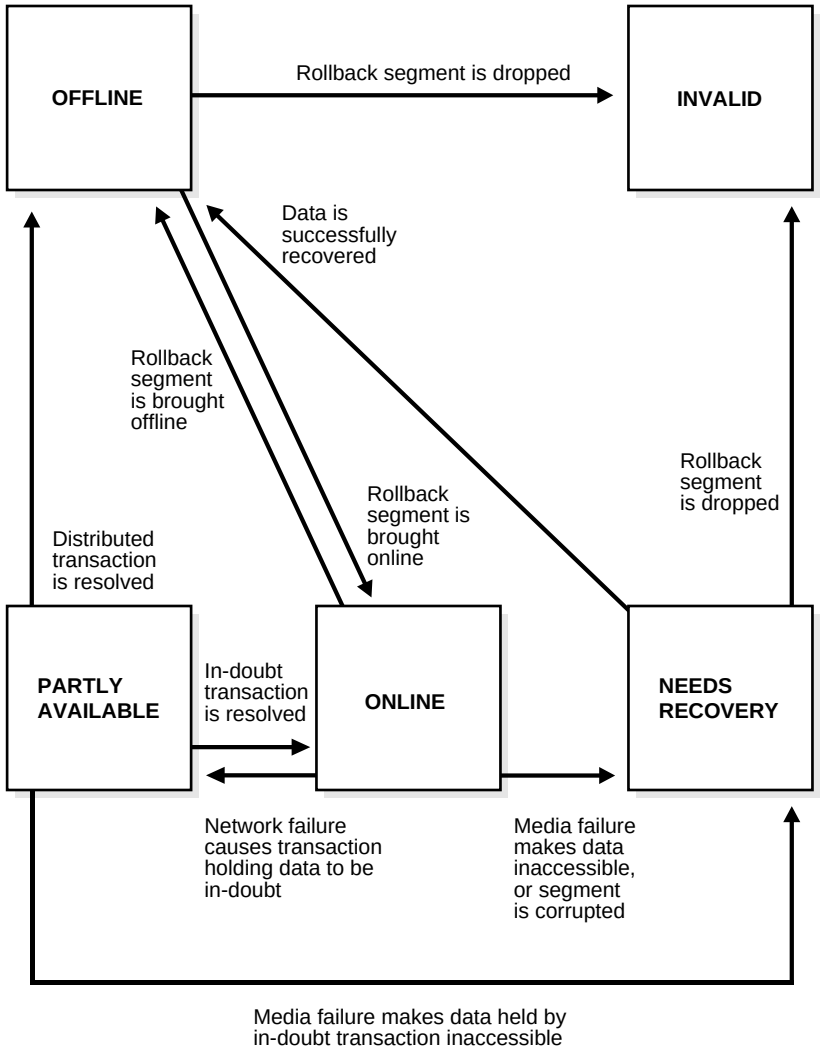
A rollback segment is always in one of several states, depending on whether it is offline, acquired by an instance, involved in an unresolved transaction, in need of recovery, or dropped. The state of the rollback segment determines whether it can be used in transactions, as well as which administrative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

The rollback segment states are:

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

The data dictionary table DBA_ROLLBACK_SEGS lists the state of each rollback segment, along with other rollback information. Figure 2–9 shows how a rollback segment moves from one state to another.

Figure 2–9 Rollback Segment States and State Transitions



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

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

> **Additional Information:** See *Oracle8 Distributed Database Systems* 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 2-18 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.

Deferred Rollback Segments

When a tablespace goes offline so that transactions cannot be rolled back immediately, Oracle writes to a *deferred rollback segment*. The deferred rollback segment contains the rollback entries that could not be applied to the tablespace, so that they can be applied when the tablespace comes back online. These segments disappear as soon as the tablespace is brought back online and recovered. Oracle automatically creates deferred rollback segments in the SYSTEM tablespace.

3

Tablespaces and Datafiles

Space — the final frontier . . .

Gene Roddenberry: *Star Trek*

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

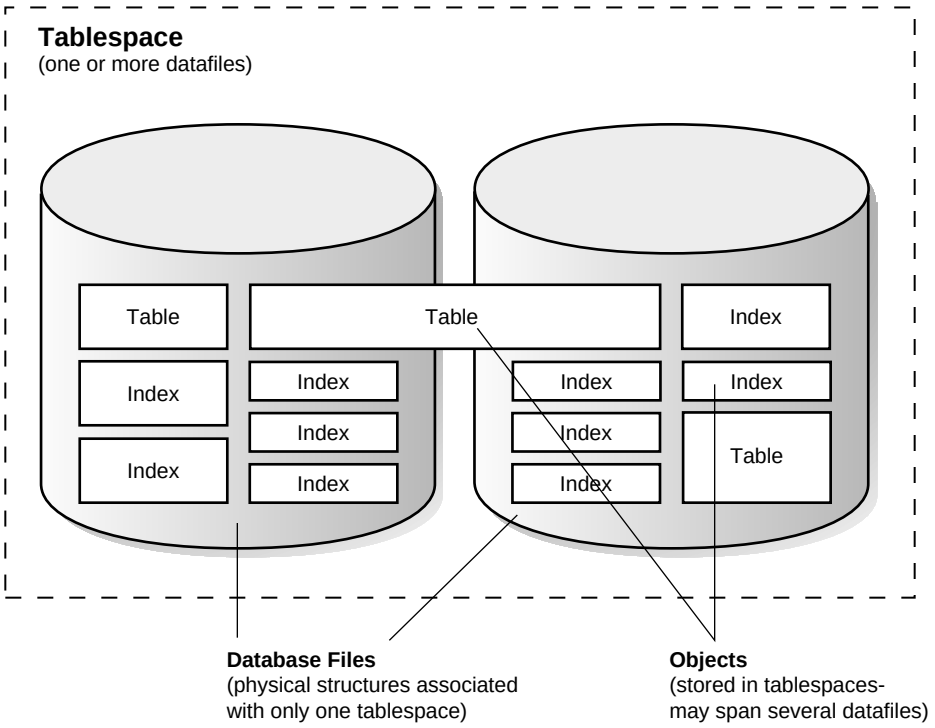
- An Introduction to Tablespaces and Datafiles
- Tablespaces
- Datafiles

> **Additional Information:** If you are using Trusted Oracle, see your *Trusted Oracle* documentation 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 3–1 illustrates this relationship.

Figure 3–1 Datafiles and Tablespaces



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

databases and tablespaces

An Oracle database consists of one or more logical storage units called tablespaces. The database's data is collectively stored in the database's tablespaces.

tablespaces and datafiles	Each tablespace in an Oracle database consists of one or more files called datafiles. These are physical structures that conform with the operating system in which Oracle is running.
databases and datafiles	A database's data is collectively stored in the datafiles that constitute each tablespace of the database. For example, the simplest Oracle database would have one tablespace and one datafile. A more complicated database might have three tablespaces, each consisting of two datafiles (for a total of six datafiles).

The following sections further explain tablespaces and datafiles.

Tablespaces

A database is divided into one or more logical storage units called *tablespaces*. The database administrator (DBA) uses tablespaces to:

- control disk space allocation for database data
- assign specific space quotas for database users
- control availability of data by taking individual tablespaces online or offline
- perform partial database backup or recovery operations
- allocate data storage across devices to improve performance

A DBA can create new tablespaces, add datafiles to tablespaces, set and alter default segment storage settings for segments created in a tablespace, make a tablespace read-only or read-write, make a tablespace temporary or permanent, and drop tablespaces.

This section includes the following topics:

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

Tablespaces are divided into logical units of storage called *segments*, which are discussed in detail in Chapter 2, “Data Blocks, Extents, and Segments”.

The SYSTEM Tablespace

Every Oracle database contains a tablespace named SYSTEM, which 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, Oracle Corporation recommends that you create at least one additional tablespace to store user data separate from data dictionary information. This gives you more flexibility in various database administration operations and reduces contention among dictionary objects and schema objects for the same datafiles.

Note: The SYSTEM tablespace is always online when the database is open. See “Bringing Tablespaces Online and Offline” on page 3-7.

All data stored on behalf of stored PL/SQL program units (procedures, functions, packages, and triggers) resides in the SYSTEM tablespace. If the database will contain many of these program units, the database administrator needs to allow for the space these objects use in the SYSTEM tablespace. For more information about these objects and the space that they require, see Chapter 17, “Procedures and Packages”, and Chapter 18, “Database Triggers”.

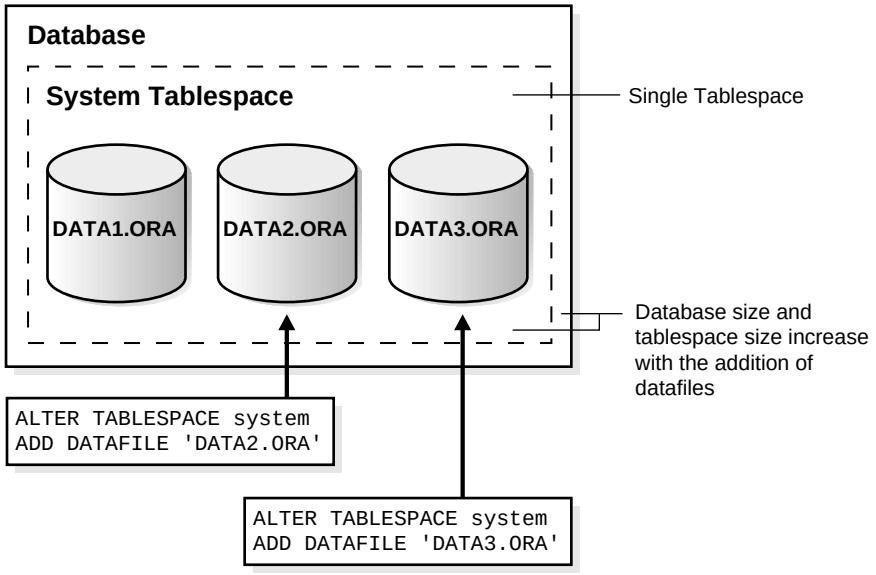
Allocating More Space for a Database

You can enlarge a database in three ways:

- add a datafile to a tablespace
- add a new tablespace
- increase the size of a datafile

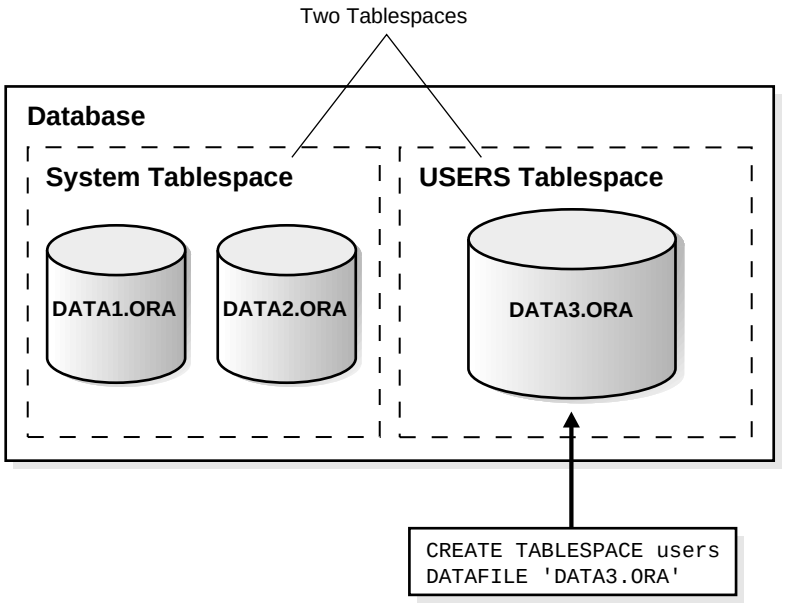
When you add another datafile to an existing tablespace, you increase the amount of disk space allocated for the corresponding tablespace. Figure 3–2 illustrates this kind of space increase.

Figure 3–2 Enlarging a Database by Adding a Datafile to a Tablespace



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

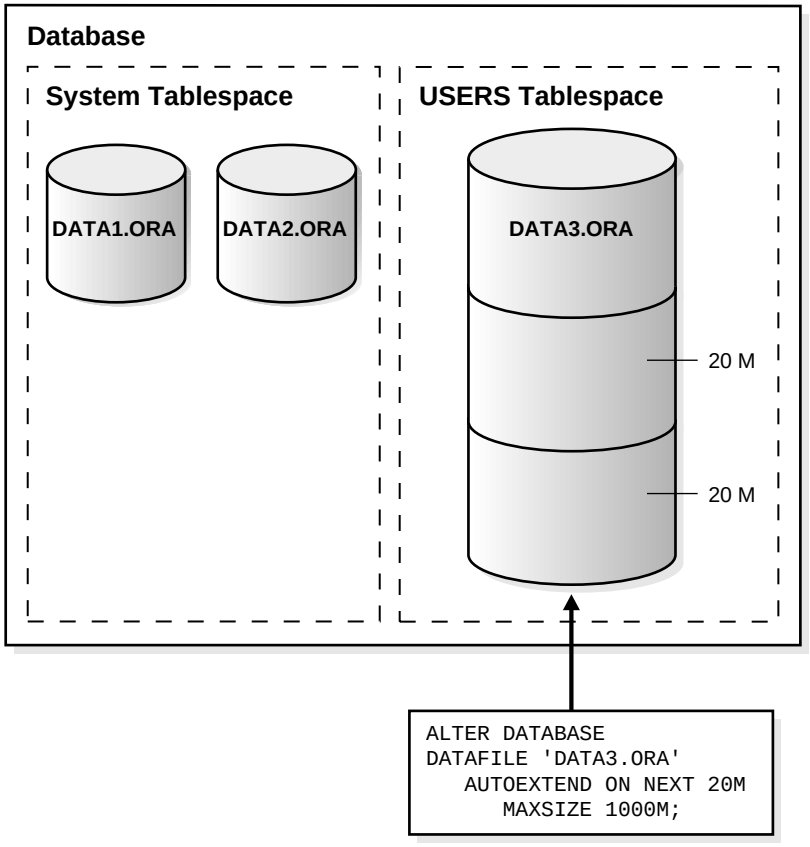
Figure 3–3 Enlarging a Database by Adding a New Tablespace



The size of a tablespace is the size of the datafile(s) that constitute the tablespace; 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 3–4 illustrates this.

Figure 3–4 Enlarging a Database by Dynamically Sizing Datafiles



Additional Information: See the *Oracle8 Administrator's Guide* for more information about increasing the amount of space in your database.

Bringing Tablespaces Online and Offline

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 is always online when the database is open because the data dictionary must always be available to Oracle.

A tablespace is normally online so that the data contained within it is available to database users. However, the database administrator might take a tablespace offline

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

You cannot take a tablespace offline if it contains any rollback segments that are in use. See "Rollback Segments" on page 2-17 for more information.

When a Tablespace Goes Offline

When a tablespace goes offline, Oracle does not permit any subsequent SQL statements to reference objects contained in that tablespace. Active transactions with completed statements that refer to data in that tablespace are not affected at the transaction level. Oracle saves rollback data corresponding to those completed statements in a deferred rollback segment (in the SYSTEM tablespace). When the tablespace is brought back online, Oracle applies the rollback data to the tablespace, if needed.

When a tablespace goes offline or comes back online, this is recorded in the data dictionary in the SYSTEM tablespace. If a tablespace was offline when you shut down a database, the tablespace remains offline when the database is subsequently mounted and reopened.

You can bring a tablespace online only in the database in which it was created because the necessary data dictionary information is maintained in the SYSTEM tablespace of that database. An offline tablespace cannot be read or edited by any utility other than Oracle. Thus, tablespaces cannot be transferred from database to database.

Additional Information: Transfer of Oracle data can be achieved with tools described in *Oracle8 Utilities.*

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

Using Tablespaces for Special Procedures

If you create multiple tablespaces to separate different types of data, you take specific tablespaces offline for various procedures; other tablespaces remain online and the information in them is still available for use. However, special circumstances can occur when tablespaces are taken offline. For example, if two tablespaces are used to separate table data from index data, the following is true:

- If the tablespace containing the indexes is offline, queries can still access table data because queries do not require an index to access the table data.
- If the tablespace containing the tables is offline, the table data in the database is not accessible because the tables are required to access the data.

In summary, if Oracle has enough information in the online tablespaces to execute a statement, it will do so. If it needs data in an offline tablespace, then it causes the statement to fail.

Read-Only Tablespaces

The primary purpose of read-only tablespaces is to eliminate the need to perform backup and recovery of large, static portions of a database. Oracle never updates the files of a read-only tablespace, and therefore the files can reside on read-only media, such as CD ROMs or WORM drives.

> **Note:** Because you can only bring a tablespace online in the database in which it was created, read-only tablespaces are not meant to satisfy archiving or data publishing requirements.

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

> **Additional Information:** See the *Oracle8 SQL Reference* for information on the ALTER TABLESPACE command.

Making a tablespace read-only does not change its offline or online status. Offline datafiles cannot be accessed. Bringing a datafile in a read-only tablespace online makes the file only readable. The file cannot be written to unless its associated tablespace is returned to the read-write state. You can take the files of a read-only tablespace online or offline independently using the DATAFILE option of the ALTER DATABASE command.

Read-only tablespaces cannot be modified. To update a read-only tablespace, you must first make the tablespace read-write. After updating the tablespace, you can then reset it to be read-only.

Because read-only tablespaces cannot be modified, they do not need repeated backup. Also, should you need to recover your database, you do not need to recover any read-only tablespaces, because they could not have been modified. However, read-only tablespaces may need attention during instance or media recovery, depending upon whether and when they have ever been read-write.

> **Additional Information:** See *Oracle8 Backup and Recovery Guide* for more information about recovery.

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

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

Temporary Tablespaces

You can manage space for sort operations more efficiently by designating *temporary tablespaces* exclusively for sorts. Doing so effectively eliminates serialization of space management operations involved in the allocation and deallocation of sort space. All operations that use sorts — including joins, index builds, ordering (ORDER BY), the computation of aggregates (GROUP BY), and the ANALYZE command to collect optimizer statistics — benefit from temporary tablespaces. The performance gains are significant in Oracle Parallel Server environments.

A *temporary tablespace* can be used only for sort segments. (It is not the same as a tablespace that a user designates for temporary segments, which can be any tablespace available to the user.) No permanent 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 expands by allocating extents until the segment size is equal to or greater than the total storage demands of all of the active sorts running on that instance.

You create temporary tablespaces using the following SQL syntax:

```
CREATE TABLESPACE tablespace TEMPORARY | PERMANENT;
```

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

```
ALTER TABLESPACE tablespace TEMPORARY;
```

> **Additional Information:** See *Oracle8 SQL Reference* for more information on the CREATE TABLESPACE and ALTER TABLESPACE commands.

Datafiles

A tablespace in an Oracle database consists of one or more physical *datafiles*. A datafile can be associated with only one tablespace and only one database.

Oracle creates a datafile for a tablespace by allocating the specified amount of disk space plus the overhead required for the file header. When a datafile is created, the operating system in which Oracle is running is responsible for clearing old information and authorizations from a file before allocating it to Oracle. If the file is large, this process might take a significant amount of time.

> **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.

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

Datafile Contents

When a datafile is first created, the allocated disk space is formatted but does not contain any user data; however, Oracle reserves the space to hold the data for future segments of the associated tablespace — it is used exclusively by Oracle. As the data grows in a tablespace, Oracle uses the free space in the associated datafiles to allocate extents for the segment. See Chapter 2, “Data Blocks, Extents, and Segments”, for more information.

The data associated with schema objects in a tablespace is physically stored in one or more of the datafiles that constitute the tablespace. Note that a schema object does not correspond to a specific datafile; rather, a datafile is a repository for the data of any object within a specific tablespace. Oracle allocates space for the data associated with an object in one or more datafiles of a tablespace. Therefore, an object can “span” one or more datafiles. Unless table “striping” is used (where data is spread across more than one disk), the database administrator and end users cannot control which datafile stores 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 schema objects in the tablespace grow. This functionality enables you to have fewer datafiles per tablespace and can simplify administration of datafiles.

> **Additional Information:** See the *Oracle8 Administrator’s Guide* for more information about resizing datafiles.

Offline Datafiles

You can take tablespaces *offline* (make unavailable) or bring them *online* (make available) at any time except SYSTEM. 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 some database recovery procedures.

4

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

An Introduction to the Data Dictionary

One of the most important parts of an Oracle database is its *data dictionary*, which is a **read-only** set of tables that provides information about its associated database. A data dictionary contains:

- the definitions of all schema objects in the database (tables, views, indexes, clusters, synonyms, sequences, procedures, functions, packages, triggers, and so on)
- how much space has been allocated for, and is currently used by, the schema objects
- default values for columns
- integrity constraint information
- the names of Oracle users
- privileges and roles each user has been granted
- auditing information, such as who has accessed or updated various schema objects
- in Trusted Oracle, the labels of all schema objects and users (see your *Trusted Oracle* documentation)
- other general database information

The data dictionary is structured in tables and views, just like other database data. All the data dictionary tables and views for a given database are stored in that database's SYSTEM tablespace.

Not only is the data dictionary central to every Oracle database, it is an important tool for all users, from end users to application designers and database administrators. To access the data dictionary, you use SQL statements. Because the data dictionary is read-only, you can issue only queries (SELECT statements) against the tables and views of the data dictionary.

The Structure of the Data Dictionary

A database's data dictionary consists of:

base tables	The underlying tables that store information about the associated database. Only Oracle should write to and read these tables. Users rarely access them directly because they are normalized, and most of the data is stored in a cryptic format.

user-accessible views — The views that summarize and display the information stored in the base tables of the data dictionary. These views decode the base table data into useful information, such as user or table names, using joins and WHERE clauses to simplify the information. Most users are given access to the views rather than the base tables.

SYS, the Owner of the Data Dictionary

The Oracle user SYS owns all base tables and user-accessible views of the data dictionary. Therefore, no Oracle user should **ever** alter (update, delete, or insert) any rows or schema objects contained in the SYS schema, because such activity can compromise data integrity. The security administrator should keep strict control of this central account.

WARNING: Altering or manipulating the data in underlying data dictionary tables can permanently and detrimentally affect the operation of a database.

How the Data Dictionary Is Used

The data dictionary has three primary uses:

- Oracle accesses the data dictionary to find information about users, schema objects, and storage structures.
- Oracle modifies the data dictionary every time that a data definition language (DDL) statement is issued.
- Any Oracle user can use the data dictionary as a read-only reference for information about the database.

How Oracle Uses the Data Dictionary

Data in the base tables of the data dictionary **is necessary for Oracle to function.** Therefore, only Oracle should write or change data dictionary information.

During database operation, Oracle reads the data dictionary to ascertain that schema objects exist and that users have proper access to them. Oracle also updates the data dictionary continuously to reflect changes in database structures, auditing, grants, and data.

For example, if user KATHY creates a table named PARTS, new rows are added to the data dictionary that reflect the new table, columns, segment, extents, and the privileges that KATHY has on the table. This new information is then visible the next time the dictionary views are queried.

Public Synonyms for Data Dictionary Views

Oracle creates public synonyms on many data dictionary views so that users can access them conveniently. (The security administrator can also create additional public synonyms for schema objects that are used systemwide.) Users should avoid naming their own schema objects with the same names as those used for public synonyms.

Caching of the Data Dictionary for Fast Access

Much of the data dictionary information is cached in the SGA (the *dictionary cache*), because Oracle constantly accesses the data dictionary during database operation to validate user access and to verify the state of schema objects. All information is stored in memory using the LRU (*least recently used*) algorithm.

Information typically kept in the caches is that required for parsing. The COMMENTS columns describing the tables and their columns are not cached unless they are accessed frequently.

Other Programs and the Data Dictionary

Other Oracle products can reference existing views and create additional data dictionary tables or views of their own. Application developers who write programs that refer to the data dictionary should refer to the public synonyms rather than the underlying tables: the synonyms are less likely to change between software releases.

Adding New Data Dictionary Items

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

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

Deleting Data Dictionary Items

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

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

How Oracle Users Can Use the Data Dictionary

The views of the data dictionary serve as a reference for all database users. You access the data dictionary views via the SQL language. Some views are accessible to all Oracle users; others are intended for administrators only.

The data dictionary is always available when the database is open. It resides in the SYSTEM tablespace, which is always online.

The data dictionary consists of sets of views. In many cases, a set consists of three views containing similar information and distinguished from each other by their prefixes:

Table 4–1 Data Dictionary View Prefixes

Prefix	Scope
USER	user's view (what is in the user's schema)
ALL	expanded user's view (what the user can access)
DBA	database administrator's view (what all users can access)

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.

> **Additional Information:** See the *Oracle8 Reference* for a complete list of data dictionary views and their columns.

Views with the Prefix USER

The views most likely to be of interest to typical database users are those with the prefix USER. These views

- refer to the user's own private environment in the database, including information about schema objects created by the user, grants made by the user, and so on
- display only rows pertinent to the user
- have columns identical to the other views, except that the column OWNER is implied (the current user)
- return a subset of the information in the ALL_ views
- can have abbreviated PUBLIC synonyms for convenience

For example, the following query returns all the objects contained in your schema:

```
SELECT object_name, object_type FROM user_objects;
```

Views with the Prefix ALL

Views with the prefix ALL refer to the user's overall perspective of the database. These views return information about schema objects to which the user has access via public or explicit grants of privileges and roles, in addition to schema objects that the user owns. For example, the following query returns information about all the objects to which you have access:

```
SELECT owner, object_name, object_type FROM all_objects;
```

Views with the Prefix DBA

Views with the prefix DBA show a global view of the entire database. Therefore, they are meant to be queried only by database administrators. Any user granted the system privilege SELECT ANY TABLE can query the DBA-prefixed views of the data dictionary.

Synonyms are not created for these views, because the DBA views should be queried only by administrators. Therefore, to query the DBA views, administrators must prefix the view name with its owner, SYS, as in

```
SELECT owner, object_name, object_type FROM sys.dba_objects;
```

Administrators can run the script file DBA_SYNONYMS.SQL to create private synonyms for the DBA views in their accounts if they have the SELECT ANY TABLE system privilege. Executing this script creates synonyms for the current user only.

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 called DUMMY and one row containing the value "X".

> **Additional Information:** See the description of the SELECT command in the *Oracle8 SQL Reference* for more information about the DUAL table.

The Dynamic Performance Tables

Throughout its operation, Oracle maintains a set of "virtual" tables that record current database activity. These tables are called *dynamic performance tables.*

Dynamic performance tables are not true tables, and they should not be accessed by most users. However, database administrators can query and create views on the tables and grant access to those views to other users.

SYS owns the dynamic performance tables; their names all begin with V_$. Views are created on these tables, and then public synonyms are created for the views. The synonym names begin with V$. For example, V$DATAFILE contains information about the database's datafiles and V$FIXED_TABLE contains information about all of the dynamic performance tables and views in the database.

> **Additional Information:** See the *Oracle8 Reference* for a complete list of the dynamic performance views' synonyms and their columns.

Part III

The Oracle Instance

Part III describes the memory structures and processes that make up an Oracle server instance and explains how the Oracle instance starts up and shuts down.

Part III contains the following chapters:

- Chapter 5, “Database and Instance Startup and Shutdown”
- Chapter 6, “Memory Structures”
- Chapter 7, “Process Structure”

5

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 procedures involved in starting and stopping an Oracle instance and database. It includes:

- Overview of an Oracle Instance
  - Connecting with Administrator Privileges
  - Parameter Files
- Instance and Database Startup
- Database and Instance Shutdown

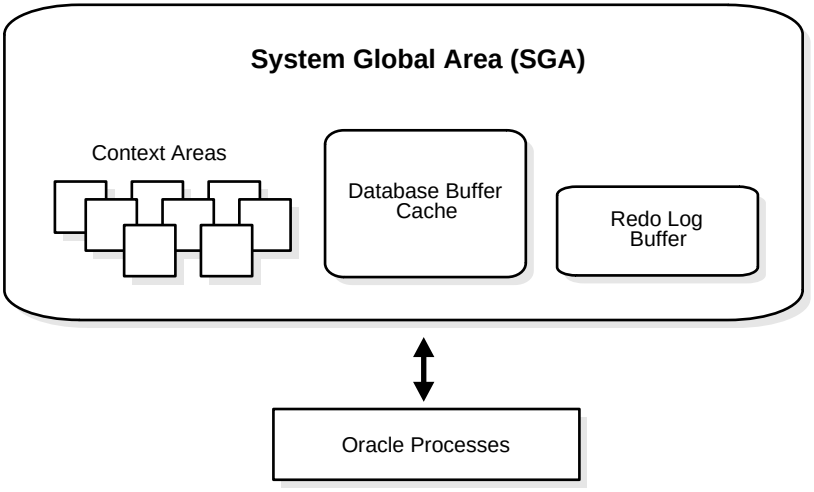
> **Additional Information:** If you are using Trusted Oracle, refer to your *Trusted Oracle* documentation for information about starting up and shutting down in that environment.

Overview of an Oracle Instance

Every running Oracle database is associated with an Oracle instance. When a database is started on a database server (regardless of the type of computer), Oracle allocates a memory area called the System Global Area (SGA) and starts one or more Oracle processes. This combination of the SGA and the Oracle processes is called an *Oracle instance*. The memory and processes of an instance manage the associated database's data efficiently and serve the one or multiple users of the database.

Figure 5–1 shows an Oracle instance. Also see Chapter 6, "Memory Structures" and Chapter 7, "Process Structure" for details about the SGA and Oracle processes.

Figure 5–1 An Oracle Instance



The Instance and the Database

After starting an instance, Oracle associates the instance with the specified database. This is called *mounting* the database. The database is then ready to be *opened*, which makes it accessible to authorized users.

Multiple instances can execute concurrently on the same computer, each accessing its own physical database. In clustered and massively parallel systems (MPP), the Oracle Parallel Server allows multiple instances to mount a single database.

> **Additional Information:** See *Oracle8 Parallel Server Concepts and Administration* for information about the Oracle Parallel Server.
>
> If you are using Trusted Oracle, see your *Trusted Oracle* documentation for information about instances in that environment.

Only the database administrator can start up an instance and open the database. If a database is open, the database administrator can shut down the database so that it is closed. When a database is *closed*, users cannot access the information that it contains.

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

Connecting with Administrator Privileges

Database startup and shutdown are powerful administrative options and are restricted to users who connect to Oracle with administrator privileges. Depending on the operating system**,** one of the following conditions establishes administrator privileges for a user:

- The user's operating system privileges allow him or her to connect using administrator privileges.
- The user is granted the SYSDBA or SYSOPER privileges and the database uses password files to authenticate database administrators.
- The database has a password for the INTERNAL login, and the user knows the password.

For additional security, users who connect with administrator privileges can only connect to dedicated servers (not shared servers).

When you connect with administrator privileges, you are placed in the schema owned by SYS. This gives you access to all the objects in the SYS schema.

For more information about password files and authentication schemes for database administrators, see Chapter 25, "Controlling Database Access".

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

Parameter Files

To start an instance, Oracle must read a *parameter file* — a text file containing a list of configuration parameters (*initialization parameters*) for that instance and database. You set these parameters to particular values to initialize many of the memory and process settings of an Oracle instance. Most initialization parameters belong to one of the following groups:

- parameters that name things (such as files)
- parameters that set limits (such as maximums)
- parameters that affect capacity (such as the size of the SGA), which are called *variable parameters*

Among other things, the initialization parameters tell Oracle:

- the name of the database for which to start up an instance
- how much memory to use for memory structures in the SGA
- what to do with filled online redo log files
- the names and locations of the database's control files
- the names of private rollback segments in the database

An Example of a Parameter File

The following is an example of a typical parameter file:

```
db_block_buffers = 550
db_name = ORA8PROD
db_domain = US.ACME.COM
#
license_max_users = 64
#
control_files = filename1, filename2
#
log_archive_dest = c:\logarch
log_archive_format = arch%S.ora
log_archive_start = TRUE
log_buffer = 64512
log_checkpoint_interval = 256000
# rollback_segments = rs_one, rs_two
```

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

Changing Parameter Values

The database administrator can adjust variable parameters to improve the performance of a database system. Exactly which parameters most affect a system is a function of numerous database characteristics and variables.

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

> **Additional Information:** For descriptions of all initialization parameters, see *Oracle8 Reference*. For information about parameters that affect the SGA, see "Size of the SGA" on page 6-11.

Instance and Database Startup

The three steps to starting a Oracle database and making it available for system-wide use are:

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

A database administrator can perform these steps using Oracle Enterprise Manager.

> **Additional Information:** See *Oracle Enterprise Manager Administrator's Guide*.

Starting an Instance

When Oracle starts an instance, first it reads a parameter file to determine the values of initialization parameters and then it allocates an SGA — a shared area of memory used for database information — and creates background processes. At this point, no database is associated with these memory structures and processes.

See Chapter 6, "Memory Structures", for information about the SGA and Chapter 7, "Process Structure", for information about background processes.

Restricted Mode of Instance Startup

You can start an instance in restricted mode (or later alter an existing instance to be in restricted mode). This restricts connections to only those users who have been granted the RESTRICTED SESSION system privilege.

Forcing an Instance to Startup in Abnormal Situations

In unusual circumstances, a previous instance might not have been shut down "cleanly", for example, one of the instance's processes might not have terminated properly. In such situations, the database might return an error during normal instance startup. To resolve this problem, you must terminate all remnant Oracle processes of the previous instance before starting the new instance.

Mounting a Database

The instance mounts a database to associate the database with that instance. After mounting the database, the instance finds the database control files and opens them. (Control files are specified in the CONTROL_FILES initialization parameter in the parameter file used to start the instance.) Oracle then reads the control files to get the names of the database's datafiles and redo log files.

At this point, the database is still closed and is accessible only to the database administrator. The database administrator can keep the database closed while completing specific maintenance operations. However, the database is not yet available for normal operations.

Modes of Mounting a Database with the Parallel Server

> **Attention:** The features described in this chapter are available only if you have purchased Oracle8 Enterprise Edition with the Parallel Server Option. See *Getting to Know Oracle8 and the Oracle8 Enterprise Edition* for more information.

If Oracle allows multiple instances to mount the same database concurrently, the database administrator can choose whether to run the database in exclusive or shared 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 allow an instance to mount a database only in exclusive mode.

Shared Mode If the first instance that mounts a database is started in shared mode (also called "parallel" mode), other instances that are started in shared mode can also mount the database.

The number of instances that can mount the database is subject to a predetermined maximum, which you can specify when creating the database.

> **Additional Information:** See *Oracle8 Parallel Server Concepts and Administration* for more information about the use of multiple instances with a single database.

Opening a Database

Opening a mounted database makes it available for normal database operations. Any valid user can connect to an open database and access its information. Usually a database administrator opens the database to make it available for general use.

When you open the database, Oracle opens the online datafiles and online redo log files. If a tablespace was offline when the database was previously shut down, the tablespace and its corresponding datafiles will still be offline when you reopen the database. See "Bringing Tablespaces Online and Offline" on page 3-7.

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 last closed abnormally, either because the database administrator aborted its instance or because of a power failure, Oracle automatically performs instance recovery when the database is reopened. See "Database Instance Failure" on page 28-4.

Rollback Segment Acquisition

When you open the database, the instance attempts to acquire one or more rollback segments. See "The Rollback Segment SYSTEM" and "Oracle Instances and Types of Rollback Segments" on page 2-23.

Resolution of In-Doubt Distributed Transaction

Occasionally a database may close abnormally with one or more distributed transactions *in doubt* (neither committed nor rolled back). When you reopen the database and instance recovery is complete, the RECO background process automatically, immediately, and consistently resolves any in-doubt distributed transactions. For more information, see Chapter 30, "Distributed Databases".

> **Additional Information:** See *Oracle8 Distributed Database Systems* for information on recovery from distributed transaction failures.

Database and Instance Shutdown

The three steps to shutting down a database and its associated instance are:

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

A database administrator can perform these steps using Oracle Enterprise Manager. Oracle automatically performs all three steps whenever an instance is shut down.

> **Additional Information:** See *Oracle Enterprise Manager Administrator's Guide*.

Closing a Database

When you close a database, Oracle writes all database data and recovery data in the SGA to the datafiles and redo log files, respectively. Next, Oracle closes all online datafiles and online redo log files. (Any offline datafiles of any offline tablespaces will have been closed already. If you subsequently reopen the database, any tablespace that was offline and its datafiles remain offline and closed, respectively.) At this point, the database is closed and inaccessible for normal operations. The control files remain open after a database is closed but still mounted.

Closing the Database by Aborting the Instance

In rare emergency situations, you can abort the instance of an open database to close and completely shut down the database instantaneously. This process is fast, because the operation of writing all data in the buffers of the SGA to the datafiles and redo log files is skipped. The subsequent reopening of the database requires 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

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

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

Shutting Down an Instance

The final step in database shutdown is shutting down the instance. When you shut down an instance, the SGA is removed from memory and the background processes are terminated.

Abnormal Instance Shutdown

In unusual circumstances, shutdown of an instance might not occur cleanly; all memory structures might not be removed from memory or one of the background processes might not be terminated. When remnants of a previous instance exist, subsequent instance startup most likely will fail. In such situations, the database administrator can force the new instance to start up by first removing the remnants of the previous instance and then starting a new instance, or by issuing a SHUTDOWN ABORT command in Oracle Enterprise Manager.

> **Additional Information:** For more detailed information on instance and database startup and shutdown, see *Oracle8 Administrator's Guide*.

6

Memory Structures

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:

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

Introduction to Oracle Memory Structures

Oracle uses memory to store various information:

- program code being executed
- information about a connected session, even if it is not currently active
- information needed during program execution (for example, the current state of a query from which rows are being fetched)
- information that is shared and communicated among Oracle processes (for example, locking information)
- cached data that is also permanently stored on peripheral memory (for example, data blocks and redo log entries)

The basic memory structures associated with Oracle include:

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

System Global Area (SGA)

A system global area (SGA) is a group of shared memory structures that contain data and control information for one Oracle database instance. If multiple users are concurrently connected to the same instance, the data in the instance's SGA is "shared" among the users. Consequently, the SGA is sometimes referred to as the "shared global area".

As described in "Overview of an Oracle Instance" on page 5-2, an SGA and Oracle processes constitute an Oracle instance. Oracle automatically allocates memory for an SGA when you start an instance and the operating system reclaims the memory when you shut down the instance. Each instance has its own SGA.

The SGA is read-write; all users connected to a multiple-process database instance may read information contained within the instance's SGA, and several processes write to the SGA during execution of Oracle.

The SGA contains the following data structures:

- the database buffer cache
- the redo log buffer
- the shared pool
- the data dictionary cache
- other miscellaneous information

Part of the SGA contains general information about the state of the database and the instance, which the background processes need to access; this is called the *fixed SGA*. No user data is stored here.

The SGA also includes information communicated between processes, such as locking information.

If the system uses multithreaded server architecture the request and response queues, and some contents of the program global areas, are in the SGA. (See "Program Global Areas (PGA)" on page 6-13 and "Dispatcher Request and Response Queues" on page 7-21.)

The Database Buffer Cache

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

The database buffer cache and the shared SQL cache are logically segmented into multiple sets. This organization into multiple sets reduces contention on multiprocessor systems.

Organization of the Database Buffer Cache

The buffers in the cache are organized in two lists: the dirty list and the least recently used (LRU) list. The *dirty list* holds *dirty buffers*, which contain data that has been modified but has not yet been written to disk. The *least recently used (LRU) list* holds free buffers, pinned buffers, and dirty buffers that have not yet been moved to the dirty list. *Free buffers* have not been modified and are available for use. *Pinned buffers* are currently being accessed.

When an Oracle process accesses a buffer, the process moves the buffer to the most recently used (MRU) end of the LRU list. As more buffers are continually moved to the MRU end of the LRU list, dirty buffers "age" towards the LRU end of the LRU list.

The first time an Oracle user process requires a particular piece of data, it searches for the data in the database buffer cache. If the process finds the data already in the cache (a *cache hit*), it can read the data directly from memory. If the process cannot find the data in the cache (a *cache miss*), it must copy the data block from a datafile on disk into a buffer in the cache before accessing the data. Accessing data through a cache hit is faster than data access through a cache miss.

Before reading a data block into the cache, the process must first find a free buffer. The process searches the LRU list, starting at the least recently used end of the list. The process searches either until it finds a free buffer or until it has searched the threshold limit of buffers.

If the user process finds a dirty buffer as it searches the LRU list, it moves that buffer to the dirty list and continues to search. When the process finds a free buffer, it reads the data block from disk into the buffer and moves the buffer to the MRU end of the LRU list.

If an Oracle user process searches the threshold limit of buffers without finding a free buffer, the process stops searching the LRU list and signals the DBW0 background process to write some of the dirty buffers to disk. For more information about the DBW0 process (or multiple DBW*n* processes), see "Database Writer (DBWn)" on page 7-8.

The LRU Algorithm and Full Table Scans

When the user process is performing a full table scan, it reads the blocks of the table into buffers and puts them on the LRU end (instead of the MRU end) of the LRU list. This is because a fully scanned table usually is needed only briefly, so the blocks should be moved out quickly to leave more frequently used blocks in the cache.

You can control this default behavior of blocks involved in table scans on a table-by-table basis. To specify that blocks of the table are to be placed at the MRU end of the list during a full table scan, use the CACHE clause when creating or altering a table or cluster. You may want to specify this behavior for small lookup tables or large static historical tables to avoid I/O on subsequent accesses of the table.

> **Additional Information:** See *Oracle8 SQL Reference* for information on the CACHE clause.

Size of the Database Buffer Cache

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

The cache has a limited size, so not all the data on disk can fit in the cache. When the cache is full, subsequent cache misses cause Oracle to write dirty data already in the cache to disk to make room for the new data. (If a buffer is not dirty, it does not need to be written to disk before a new block can be read into the buffer.) Subsequent access to any data that was written to disk results in additional cache misses.

The size of the cache affects the likelihood that a request for data will result in a cache hit. If the cache is large, it is more likely to contain the data that is requested. Increasing the size of a cache increases the percentage of data requests that result in cache hits.

> **Additional Information:** See *Oracle8 Tuning* for more information on the buffer cache.

Multiple Buffer Pools

You can configure the database buffer cache with separate buffer pools that either keep data in the buffer cache or make the buffers available for new data immediately after using the data blocks. Particular schema objects (tables, clusters, indexes, and partitions) can then be assigned to the appropriate buffer pool to control the way their data blocks age out of the cache.

- The KEEP buffer pool retains the schema object's data blocks in memory.
- The RECYCLE buffer pool eliminates data blocks from memory as soon as they are no longer needed.
- The DEFAULT buffer pool contains data blocks from schema objects that are not assigned to any buffer pool, as well as schema objects that are explicitly assigned to the DEFAULT pool.

The initialization parameters that configure the KEEP and RECYCLE buffer pools are BUFFER_POOL_KEEP and BUFFER_POOL_RECYCLE.

> **Additional Information:** See *Oracle8 Tuning* for more information on buffer pools, and see *Oracle8 SQL Reference* for the syntax of the BUFFER_POOL option of the STORAGE clause.

The Redo Log Buffer

The *redo log buffer* is a circular buffer in the SGA that holds information about changes made to the database. This information is stored in *redo entries*. Redo entries contain the information necessary to reconstruct, or redo, changes made to the database by INSERT, UPDATE, DELETE, CREATE, ALTER, or DROP operations. Redo entries are used for database recovery, if necessary.

Redo entries are copied by Oracle server processes from the user's memory space to the redo log buffer in the SGA. The redo entries take up continuous, sequential space in the buffer. The background process LGWR writes the redo log buffer to the active online redo log file (or group of files) on disk.

> **Additional Information:** See "Log Writer Process (LGWR)" on page 7-9 for more information about how the redo log buffer is written to disk, and see *Oracle8 Backup and Recovery Guide* for information about online redo log files and groups.

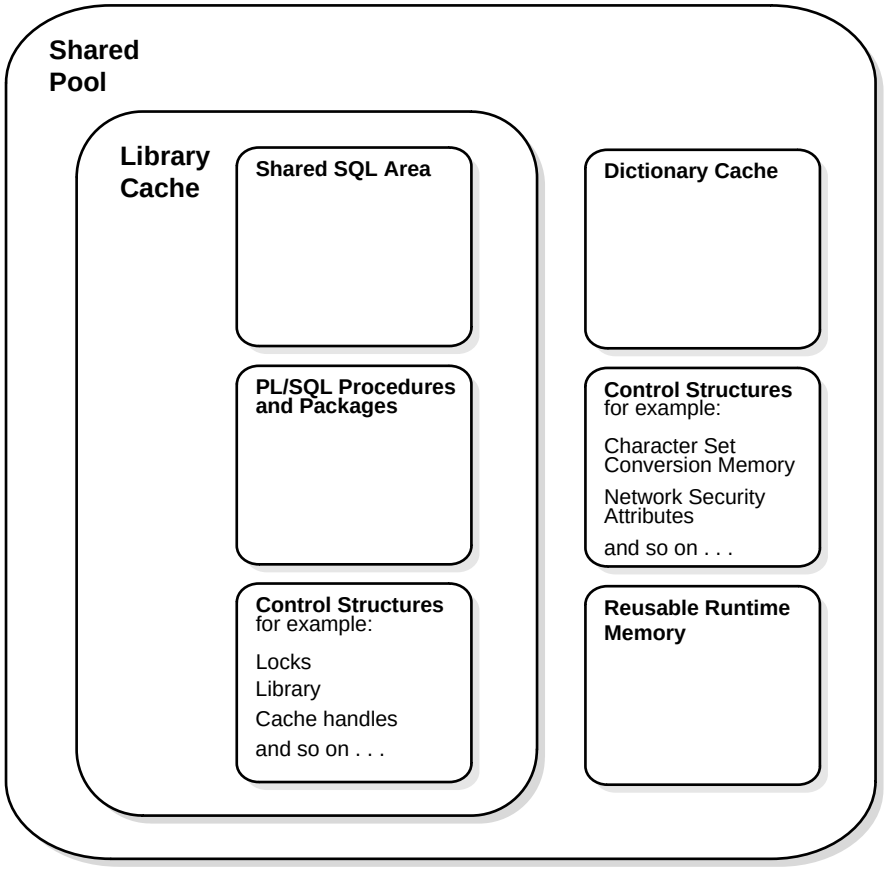
The initialization parameter LOG_BUFFER determines the size (in bytes) of the redo log buffer. In general, larger values reduce log file I/O, particularly if transactions are long or numerous. The default setting is four times the maximum data block size for the host operating system.

The Shared Pool

The shared pool portion of the SGA contains three major areas: library cache, dictionary cache, and control structures. Figure 6–1 shows the contents of the shared pool.

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

Figure 6–1 Contents of the Shared Pool



Library Cache

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

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

Shared SQL Areas and Private SQL Areas

Oracle represents each SQL statement it executes with a *shared SQL area* and a *private SQL area*. Oracle recognizes when two users are executing the same SQL statement and reuses the shared SQL area for those users. However, each user must have a separate copy of the statement's private SQL area.

Shared SQL Areas A shared SQL area contains the parse tree and execution plan for a single SQL statement, or for identical SQL statements. Oracle saves memory by using one shared SQL area for multiple identical DML statements, particularly when many users execute the same application. A shared SQL area is always in the shared pool.

> **Additional Information:** See *Oracle8 Tuning* for information about the criteria that determine identical SQL statements.

Oracle allocates memory from the shared pool when a SQL statement is parsed; the size of this memory depends on the complexity of the statement. If a SQL statement requires a new shared SQL area and the entire shared pool has already been allocated, Oracle can deallocate items from the pool using a modified least recently used algorithm until there is enough free space for the new statement's shared SQL area. If Oracle deallocates a shared SQL area, the associated SQL statement must be reparsed and reassigned to another shared SQL area when it is next executed.

Private SQL Areas A private SQL area contains data such as bind information and runtime buffers. Each session that issues a SQL statement has a private SQL area. Each user that submits an identical SQL statement has his or her own private SQL area that uses a single shared SQL area; many private SQL areas can be associated with the same shared SQL area. (See "Connections and Sessions" on page 7-4 for more information about sessions.)

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

- The *persistent area* contains bind information that persists across executions, code for datatype conversion (in case the defined datatype is not the same as the datatype of the selected column), and other state information (like recursive or remote cursor numbers or the state of a parallel query). The size of the persistent area depends on the number of binds and columns specified in the statement. For example, the persistent area is larger if many columns are specified in a query.
- The *runtime area* contains information used while the SQL statement is being executed. The size of the runtime area depends on the type and complexity of the SQL statement being executed and on the sizes of the rows that are pro-

cessed by the statement. In general, the runtime area is somewhat smaller for INSERT, UPDATE, and DELETE statements than it is for SELECT statements.

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

The location of a private SQL area depends on the type of connection established for a session. If a session is connected via a dedicated server, private SQL areas are located in the user's PGA. However, if a session is connected via the multithreaded server, the persistent areas and, for SELECT statements, the runtime areas, are kept in the SGA.

Cursors and SQL Areas The application developer of an Oracle Precompiler program or OCI program can explicitly open *cursors*, or handles to specific private SQL areas, and use them as a named resource throughout the execution of the program. Recursive cursors that Oracle issues implicitly for some SQL statements also use shared SQL areas. For more information, see "Cursors" on page 14-6.

The management of private SQL areas is the responsibility of the user process. The allocation and deallocation of private SQL areas depends largely on which application tool you are using, although the number of private SQL areas that a user process can allocate is always limited by the initialization parameter OPEN_CURSORS. The default value of this parameter is 50.

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

For queries that process large amounts of data requiring sorts, application developers should cancel the query if a partial result of a fetch is satisfactory. For example, in an Oracle Office application, a user can select from a list of over 60 templates for creating a mail message. When Oracle Office displays the first ten template names, if the user chooses one of these templates the application should cancel the processing of the rest of the query, rather than continue trying to display more template names.

PL/SQL Program Units and the Shared Pool

Oracle processes PL/SQL program units (procedures, functions, packages, anonymous blocks, and database triggers) much the same way it processes individual

SQL statements. Oracle allocates a shared area to hold the parsed, compiled form of a program unit. Oracle allocates a private area to hold values specific to the session that executes the program unit, including local, global, and package variables (also known as package instantiation) and buffers for executing SQL. If more than one user executes the same program unit, then a single, shared area is used by all users, while each user maintains a separate copy of his or her private SQL area, holding values specific to his or her session.

Individual SQL statements contained within a PL/SQL program unit are processed as described in the previous sections. Despite their origins within a PL/SQL program unit, these SQL statements use a shared area to hold their parsed representations and a private area for each session that executes the statement.

Dictionary Cache

The data dictionary is a collection of database tables and views containing reference information about the database, its structures, and its users. Oracle accesses the data dictionary frequently during the parsing of SQL statements. This access is essential to the continuing operation of Oracle. See Chapter 4, "The Data Dictionary" for more information.

The data dictionary is accessed so often by Oracle that two special locations in memory are designated to hold dictionary data. One area is called the *data dictionary cache*, also known as the *row cache* because it holds data as rows instead of buffers (which hold entire blocks of data). The other area in memory to hold dictionary data is the library cache. All Oracle user processes share these two caches for access to data dictionary information.

Allocation and Reuse of Memory in the Shared Pool

In general, any item (shared SQL area or dictionary row) in the shared pool remains until it is flushed according to a modified LRU algorithm. The memory for items that are not being used regularly is freed if space is required for new items that must be allocated some space in the shared pool. A modified LRU algorithm allows shared pool items that are used by many sessions to remain in memory as long as they are useful, even if the process that originally created the item terminates. As a result, the overhead and processing of SQL statements associated with a multiuser Oracle system is minimized.

When a SQL statement is submitted to Oracle for execution, Oracle automatically performs the following memory allocation steps:

1. Oracle checks the shared pool to see if a shared SQL area already exists for an identical statement. If so, that shared SQL area is used for the execution of the

subsequent new instances of the statement. Alternatively, if there is no shared SQL area for a statement, Oracle allocates a new shared SQL area in the shared pool. In either case, the user's private SQL area is associated with the shared SQL area that contains the statement.

> **Note:** A shared SQL area can be flushed from the shared pool, even if the shared SQL area corresponds to an open cursor that has not been used for some time. If the open cursor is subsequently used to execute its statement, Oracle reparses the statement and a new shared SQL area is allocated in the shared pool.

2. Oracle allocates a private SQL area on behalf of the session. The exact location of the private SQL area depends on the connection established for a session (see "Shared SQL Areas and Private SQL Areas" on page 6-8).

Oracle also flushes a shared SQL area from the shared pool in these circumstances:

- 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 referencing the analyzed schema object are flushed from the shared pool. The next time a flushed statement is executed, the statement is parsed in a new shared SQL area to reflect the new statistics for the schema object.
- If a schema object is referenced in a SQL statement and that object is later modified in any way, the shared SQL area is *invalidated* (marked invalid) and the statement must be reparsed the next time it is executed. See Chapter 19, "Oracle Dependency Management", 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.
- 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.

Size of the SGA

The size of the SGA is determined at instance start up. For optimal performance in most systems, the entire SGA should fit in real memory. If it does not fit in real memory and virtual memory (see "Virtual Memory" on page 6-16) is used to store parts of it, overall database system performance can decrease dramatically because

portions of the SGA are paged (written to and read from disk) by the operating system. The amount of memory dedicated to all shared areas in the SGA also has performance impact; see *Oracle8 Tuning* for more information.

The size of the SGA is determined by several initialization parameters. The parameters that most affect SGA size are:

DB_BLOCK_SIZE	The size, in bytes, of a single data block and database buffer.
DB_BLOCK_BUFFERS	The number of database buffers, each the size of DB_BLOCK_SIZE, allocated for the SGA. (The total amount of space allocated for the database buffer cache in the SGA is DB_BLOCK_SIZE times DB_BLOCK_BUFFERS.)
LOG_BUFFER	The number of bytes allocated for the redo log buffer.
SHARED_POOL_SIZE	The size in bytes of the area devoted to shared SQL and PL/SQL statements.

The memory allocated for an instance's SGA is displayed on instance startup when using Oracle Enterprise Manager (or Server Manger). You can also display the current instance's SGA size by using the Server Manager command SHOW with the SGA option.

> **Additional Information:** See the *Oracle Enterprise Manager Administrator's Guide* for more information about showing the SGA size with Oracle Enterprise Manager (or Server Manager).
>
> See *Oracle8 Tuning* for discussions of the above initialization parameters and how they affect the SGA. Also see your Oracle installation or user's guide for information specific to your operating system.

Controlling the SGA's Use of Memory

Several initialization parameters are available to control how the SGA uses memory. For details about these parameters, see *Oracle8 Reference*.

Physical Memory

The LOCK_SGA and LOCK_SGA_AREAS parameters lock the entire SGA or particular SGA areas into physical memory.

SGA Starting Address

The SHARED_MEMORY_ADDRESS and HI_SHARED_MEMORY_ADDRESS parameters specify the SGA's starting address at runtime. These parameters are used only on platforms that do not specify the SGA's starting address at link time. For 64-bit platforms, HI_SHARED_MEMORY_ADDRESS specifies the high order 32 bits of the 64-bit address.

Extended Buffer Cache Mechanism

The USE_INDIRECT_DATA_BUFFERS parameter enables the extended buffer cache mechanism for 32-bit platforms that can support more than 4 GB of physical memory.

Program Global Areas (PGA)

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

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

A PGA is allocated by Oracle when a user connects to an Oracle database and a session is created, though this varies by operating system and configuration. (See "Connections and Sessions" on page 7-4 for information about sessions.)

Contents of a PGA

The contents of a PGA vary, depending on whether the associated instance is running the multithreaded server. (See "The Multithreaded Server" on page 7-20 for more information on the multithreaded server.)

Stack Space

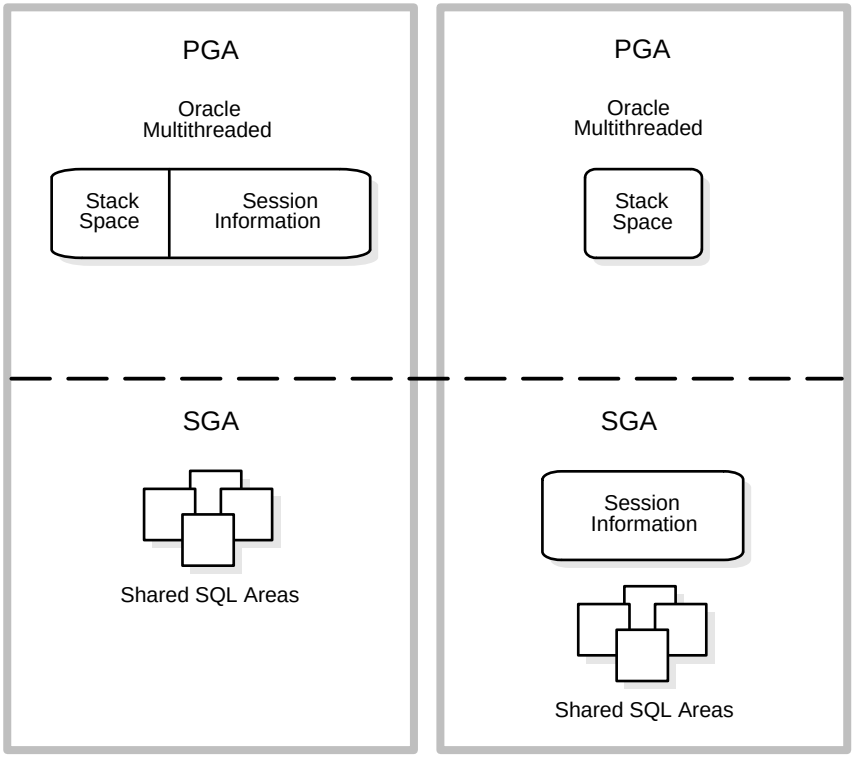
A PGA always contains a *stack space*, which is memory allocated to hold a session's variables, arrays, and other information.

Session Information

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

Figure 6–2 shows where the session information is stored in different configurations.

Figure 6–2 Location of Session Information with and without Multithreaded Server



Size of a PGA

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

The 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 DBW0 and LGWR) is affected by some additional parameters.

> **Additional Information:** See your Oracle operating-system-specific documentation for more information about the PGA.

Sort Areas

Sorting requires space in memory. Portions of memory in which Oracle sorts data are called *sort areas*. 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 default value, expressed in bytes, is operating system specific.

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

The 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.

> **Additional Information:** See *Oracle8 Tuning* for more information on sort areas and the SORT_DIRECT_WRITES parameter.

Virtual Memory

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

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

Suggestion: Usually, it is best to keep the entire SGA in real memory. On many platforms, you can lock the SGA or parts of it into real memory with the LOCK_SGA and LOCK_SGA_AREAS parameters.

Software Code Areas

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

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

Software areas are read-only and may be installed shared or nonshared. When possible, Oracle code is shared so that all Oracle users can access it without having multiple copies in memory. This results in a saving of real main memory, and improves overall performance.

User programs can be shared or nonshared. Some Oracle tools and utilities (such as SQL*Forms and SQL*Plus) can be installed shared, but some cannot. Multiple instances of Oracle can use the same Oracle code area with different databases if running on the same computer.

> **Additional Information:** The option of installing software shared is not available for all operating systems (for example, on PCs operating MS DOS). See your Oracle operating-system-specific documentation for more information.

7

Process Structure

If the good people, in their wisdom, shall see fit to keep me in the background, I have been too familiar with disappointments to be very much chagrined.

Abraham Lincoln, *Address at New Salem (1832)*

This chapter discusses the processes in an Oracle database system and the different configurations available for an Oracle system. It includes:

- Introduction to Processes
- Single-Process Oracle
- Multiple-Process Oracle
- Variations in Oracle Configuration
- Examples of How Oracle Works
- The Program Interface

Introduction to Processes

All connected Oracle users must execute two modules of code to access an Oracle database instance:

application or Oracle tool	A database user executes a database application (such as a precompiler program) or an Oracle tool (such as an Oracle Forms application or SQL*Plus), which issues SQL statements to an Oracle database.
Oracle server code	Each user has some Oracle server code executing on his or her behalf, which interprets and processes the application's SQL statements.

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

The process structure varies for different Oracle configurations, depending on the operating system and the choice of Oracle options.

Single-Process Oracle

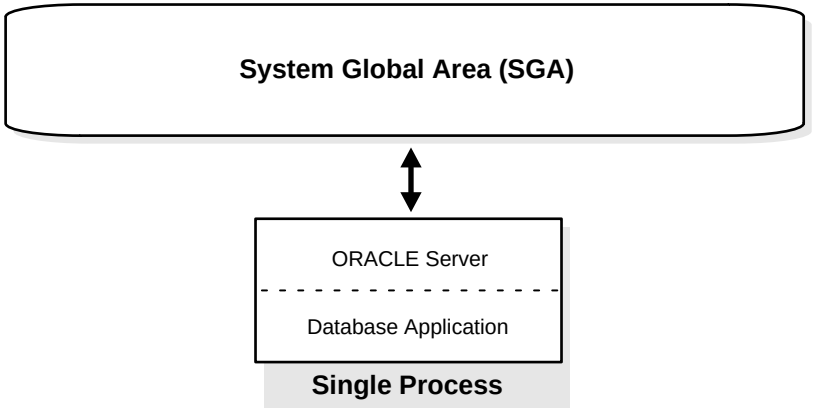
Single-process Oracle (also called *single-user Oracle*) is a database system in which one process executes all parts of the Oracle server code and the single user's application program. Different processes do not separate the execution of the Oracle instance from the client application program.

Figure 7–1 shows a single-process Oracle instance. The single process executes all code associated with the database application **and** Oracle.

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 be accessed only by a single user because MS-DOS is not capable of running multiple processes.

This configuration is not as common as multiple-process Oracle, because it doesn not take advantage of the distributed processing normally associated with database operations.

Figure 7–1 A Single-Process Oracle Instance



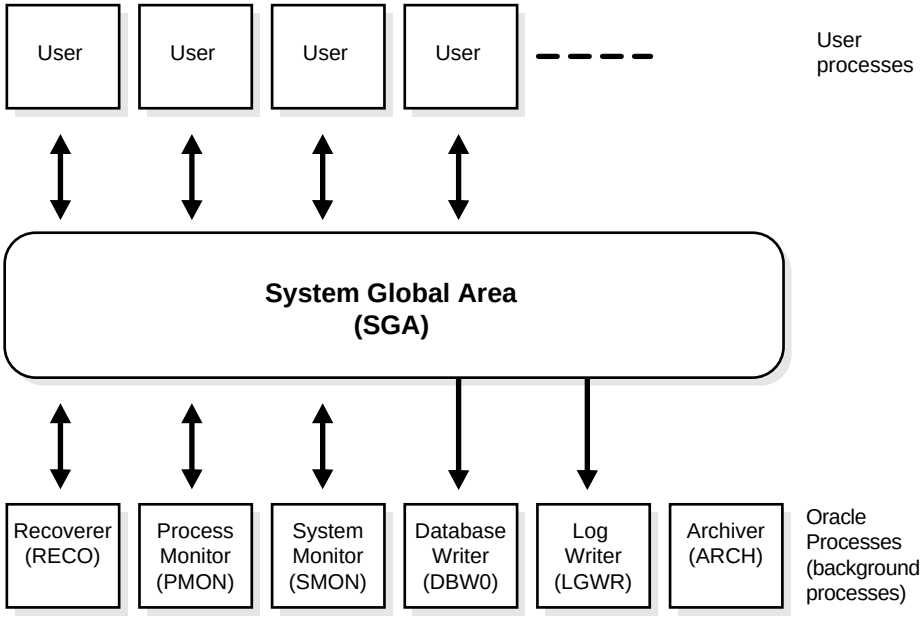
Multiple-Process Oracle

Multiple-process Oracle (also called *multiuser Oracle*) uses several processes to execute different parts of Oracle code, 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 connect to a single database instance simultaneously while the system maintains excellent performance.

Most database systems are multiuser, because one of the primary benefits of a database is managing data needed by multiple users at the same time.

Figure 7–2 illustrates a multiple-process Oracle instance. Each connected user has a separate user process, and several background processes execute Oracle. This figure might represent multiple concurrent users running an application on the same machine as Oracle; this particular configuration usually runs on a mainframe or minicomputer.

Figure 7–2 A Multiple-Process Oracle Instance



In a multiple-process system, processes can be categorized into two groups: user processes and Oracle processes. User processes execute the application or Oracle tool code, and Oracle processes execute the Oracle server code.

User Processes

When a user runs an application program (such as a Pro*C program) or an Oracle tool (such as Oracle Enterprise Manager or SQL*Plus) Oracle creates a *user process* to run the user's application.

Connections and Sessions

The terms "connection" and "session" are closely related to the term "user process", but are very different in meaning.

A *connection* is a communication pathway between a user process and an Oracle instance. A communication pathway is established using available interprocess communication mechanisms (on a computer that executes both the user process

and Oracle) or network software (when different computers execute the database application and Oracle, and communicate via a network).

A *session* is a specific connection of a user to an Oracle instance via a user process. For example, when a user starts SQL*Plus, the user must provide a valid username and password and then a session is established for that user. A session lasts from the time the user connects until the time the user disconnects or exits the database application.

Multiple sessions can be created and exist concurrently for a single Oracle user using the same username. For example, a user with the username/password of SCOTT/TIGER can connect to the same Oracle instance several times.

In configurations without the multithreaded server, Oracle creates a server process on behalf of each user session; however, with the multithreaded server, many user sessions can share a single server process. See "The Multithreaded Server" on page 7-20 for more information.

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. In some situations when the application and Oracle operate on the same machine, it is possible to combine the user process and corresponding server process into a single process to reduce system overhead. However, when the application and Oracle operate on different machines, a user process communicates with Oracle via a separate server process. See "Variations in Oracle Configuration" on page 7-16 for more information.

Server Processes

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

- Parse and execute SQL statements issued via the application.
- Read necessary data blocks from datafiles on disk into the shared database buffers of the SGA, if the blocks are not already present in the SGA.
- Return results in such a way that the application can process the information.

Background Processes

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

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.

> **Additional Information:** See your Oracle operating-system-specific documentation for details on how these processes are created.

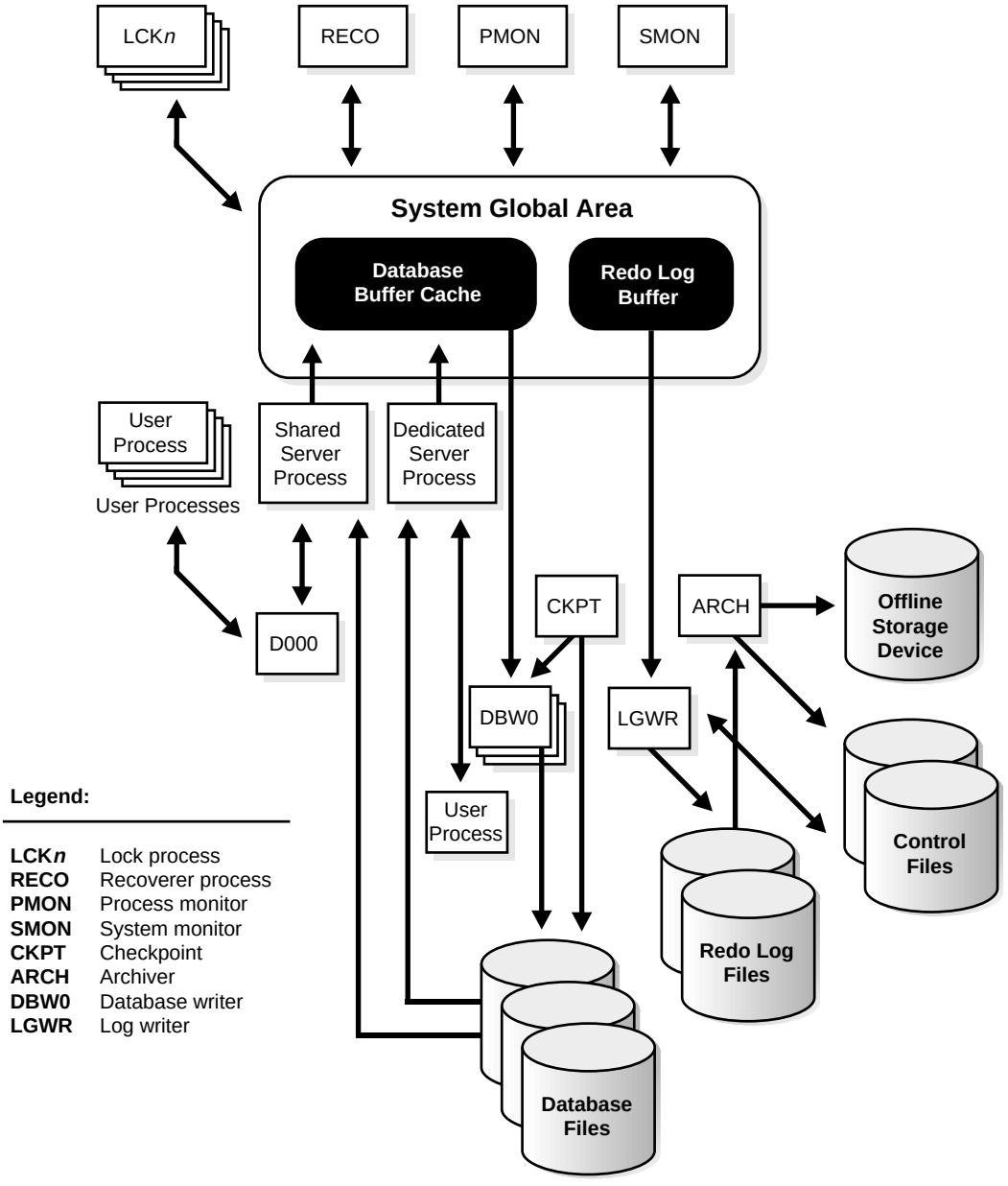
An Oracle instance may have many background processes; not all are always present. The background processes in an Oracle instance include the following:

- Database Writer (DBW0 or DBW*n*)
- Log Writer (LGWR)
- Checkpoint (CKPT)
- System Monitor (SMON)
- Process Monitor (PMON)
- Archiver (ARCH)
- Recoverer (RECO)
- Lock (LCK*n*)
- Job Queue (SNP*n*)
- Queue Monitor (QMN*n*)
- Dispatcher (D*nnn*)
- Server (S*nnn*)

Figure 7–3 illustrates how each background process interacts with the different parts of an Oracle database, and the rest of this section describes each process.

> **Additional Information:** The Oracle Parallel Server is not illustrated in Figure 7–3; see *Oracle8 Parallel Server Concepts and Administration* for more information.

Figure 7–3 The Background Processes of a Multiple-Process Oracle Instance



Database Writer (DBW*n*) The *database writer process (DBWn)* writes the contents of buffers to datafiles. The DBW*n* processes are responsible for writing modified (dirty) buffers in the database buffer cache to disk. (See "The Database Buffer Cache" on page 6-3.) Although one database writer process (DBW0) is adequate for most systems, you can configure additional processes (DBW1 through DBW9) to improve write performance if your system modifies data heavily. These additional DBW*n* processes are not useful on uniprocessor systems.

When a buffer in the database buffer cache is modified, it is marked "dirty". The primary job of the DBW*n* process is to keep the buffer cache "clean" by writing dirty buffers to disk. As buffers are dirtied by user processes, the number of free buffers diminishes. If the number of free buffers drops too low, user processes that must read blocks from disk into the cache are not able to find free buffers. DBW*n* manages the buffer cache so that user processes can always find free buffers.

The DBW*n* process writes the least recently used (LRU) buffers to disk. By writing the least recently used dirty buffers to disk, DBW*n* improves the performance of finding free buffers while keeping recently used buffers resident in memory. For example, blocks that are part of frequently accessed small tables or indexes are kept in the cache so that they do not need to be read in again from disk. The LRU algorithm keeps more frequently accessed blocks in the buffer cache so that when a buffer is written to disk, it is unlikely to contain data that may be useful soon.

The initialization parameter DB_WRITER_PROCESSES specifies the number of DBW*n* processes. If your system uses multiple DBW*n* processes, you should adjust the value of the DB_BLOCK_LRU_LATCHES parameter so that each DBW*n* process has the same number of latches (LRU buffer lists).

> **Additional Information:** See *Oracle8 Tuning* for advice on setting DB_WRITER_PROCESSES and DB_BLOCK_LRU_LATCHES.

The DBW*n* process writes dirty buffers to disk under the following conditions:

- When a server process cannot find a clean reusable buffer after scanning a threshold number of buffers, it signals DBW*n* to write. DBW*n* writes dirty buffers to disk with a single *multiblock write*. (The number of blocks written in a multiblock write varies by operating system.)
- When a *checkpoint* occurs, the log writer process (LGWR) signals DBW*n* and DBW*n* writes all buffers that need to be written for the checkpoint to complete. Checkpoints occur at each log switch, when LGWR stops writing to one online redo log file and starts writing to another, and at regular intervals specified by the initialization parameters LOG_CHECKPOINT_INTERVAL and LOG_CHECKPOINT_TIMEOUT.

- DBW*n* periodically writes buffers to advance the position in the redo thread (log) from which crash or instance recovery needs to begin *(incremental checkpointing)*. This log position is determined by the oldest dirty buffer in the buffer cache. Incremental checkpointing occurs when the number of dirty buffers in the cache is greater than a threshold specified by the initialization parameter DB_BLOCK_MAX_DIRTY_TARGET. See "Incremental Checkpointing" on page 28-4 for more information.

In all cases, DBW*n* performs batched (multiblock) writes to improve efficiency.

On some platforms, the DBW0 process can have multiple I/O server processes. If one of these processes blocks during a write to one disk, the others can continue writing to other disks. These processes cannot be used on systems with multiple DBW*n* processes. The initialization parameter DBWR_IO_SLAVES controls the number of I/O server processes.

> **Additional Information:** See your Oracle operating system-specific documentation for information about multiple I/O server processes on your platform.
>
> Also see *Oracle8 Tuning* for information about how to monitor and tune the performance of a single DBW0 process or multiple DBW*n* processes.

Log Writer Process (LGWR) The *log writer process (LGWR)* is responsible for redo log buffer management — writing the redo log buffer to a redo log file on disk (see "The Redo Log Buffer" on page 6-6). LGWR writes all redo entries that have been copied into the buffer since the last time it wrote.

The redo log buffer is a circular buffer. When LGWR writes redo entries from the redo log buffer to a redo log file, server processes can then copy new entries over the entries in the redo log buffer that have been written to disk. LGWR normally writes fast enough to ensure that space is always available in the buffer for new entries, even when access to the redo log is heavy.

LGWR writes one contiguous portion of the buffer to disk. LGWR writes:

- a commit record when a user process commits a transaction
- redo log buffers
  - every three seconds
  - when the redo log buffer is one-third full
  - when a DBW*n* process writes modified buffers to disk, if necessary

> **Note:** Before DBW*n* can write a modified buffer, all redo records associated with the changes to the buffer must be written to disk (the *write-ahead protocol*). If DBW*n* finds that some redo records have not been written, it signals LGWR to write the redo records to disk and waits for LGWR to complete writing the redo log buffer before it can write out the data buffers.

LGWR writes synchronously to the active mirrored group of online redo log files. If one of the files in the group is damaged or unavailable, LGWR continues writing to other files in the group and logs an error in the LGWR trace file and in the system ALERT file (see "Trace Files and the ALERT File" on page 7-14). If all files in a group are damaged, or the group is unavailable because it has not been archived, LGWR cannot continue to function.

When a user issues a COMMIT statement, LGWR puts a commit record in the redo log buffer and writes it to disk immediately, along with the transaction's redo entries. The corresponding changes to data blocks are deferred until it is more efficient to write them. This is called a "fast commit" mechanism. The atomic write of the redo entry containing the transaction's commit record is the single event that determines the transaction has committed. Oracle returns a success code to the committing transaction, even though the data buffers have not yet been written to disk.

> **Note:** Sometimes, if more buffer space is needed, LGWR writes redo log entries before a transaction is committed. These entries become permanent only if the transaction is later committed.

When a user commits a transaction, the transaction is assigned a *system change number (SCN),* which Oracle records along with the transaction's redo entries in the redo log. SCNs are recorded in the redo log so that recovery operations can be synchronized in Oracle Parallel Server configurations and distributed databases.

> **Additional Information:** See *Oracle8 Parallel Server Concepts and Administration* and the *Oracle8 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 and as this happens, other users issue COMMIT statements. However, LGWR cannot write to the online redo log file to

commit these transactions until it has completed its previous write operation. After the first transaction's entries are written to the online redo log file, the entire list of redo entries of waiting transactions (not yet committed) can be written to disk in one operation, requiring less I/O than would transaction entries handled individually. Therefore, Oracle minimizes disk I/O and maximizes performance of LGWR. If requests to commit continue at a high rate, then every write (by LGWR) from the redo log buffer may contain multiple commit records.

On some platforms, the LGWR process can have multiple I/O server processes. If one of these processes blocks during a write to one disk, the others can continue writing to other disks. The initialization parameter LGWR_IO_SLAVES controls the number of I/O server processes.

> **Additional Information:** See your Oracle operating system-specific documentation for information about multiple I/O server processes on your platform.
>
> For information about how to monitor and tune the performance of LGWR, see *Oracle8 Tuning.*

Checkpoint Process (CKPT) When a checkpoint occurs, Oracle must update the headers of all datafiles to record the details of the checkpoint. This is done by the CKPT process. The CKPT process does not write blocks to disk; DBW*n* always performs that work.

The statistic *DBWR checkpoints* displayed by the System_Statistics monitor in Oracle Enterprise Manager indicates the number of checkpoint requests completed.

> **Additional Information:** See the *Oracle8 Administrator's Guide* for information about the effects of changing the checkpoint interval.
>
> See *Oracle8 Parallel Server Concepts and Administration* for information about CKPT in an Oracle Parallel Server.

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 and for coalescing contiguous free extents to make larger blocks of free space available. SMON "wakes up" regularly to check whether it is needed. Other processes can call SMON if they detect a need for SMON to wake up.

In an Oracle Parallel Server environment, SMON also performs instance recovery for a failed CPU or instance.

> **Additional Information:** See *Oracle8 Parallel Server Concepts and Administration* for more information about SMON in an Oracle Parallel Server.

Process Monitor (PMON) The *process monitor (PMON)* performs process recovery when a user process fails. PMON is responsible for cleaning up the database buffer cache and freeing resources that the user process was using. For example, it resets the status of the active transaction table, releases locks, and removes the process ID from the list of active processes.

PMON also periodically checks the status of dispatcher and server processes, and restarts any that have died (but not any that Oracle has terminated 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 Process (RECO) The *recoverer process (RECO)* is a background process used with the distributed database configuration that automatically resolves failures involving distributed transactions.

The RECO process of a node automatically connects to other databases involved in an in-doubt distributed transaction. When the RECO process reestablishes a connection between involved database servers, it automatically resolves all in-doubt transactions, removing from each database's pending transaction table any rows that correspond to the resolved in-doubt transactions.

If the RECO process fails to connect with a remote server, RECO automatically tries to connect again after a timed interval. However, RECO waits an increasing amount of time (growing exponentially) before it attempts another connection.

> **Additional Information:** For more information about distributed transaction recovery, see *Oracle8 Distributed Database Systems*.

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

Archiver Process (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.

> **Additional Information:** For information on archiving the online redo log, see "The Redo Log" on page 28-7 and *Oracle8 Backup and Recovery Guide*.
>
> See your Oracle operating system-specific documentation for details of using the ARCH process.

Lock Processes (LCK*n*) With the Parallel Server option, up to ten *lock processes* (LCK0, . . ., LCK9) provide interinstance locking. A single LCK process (LCK0) is sufficient for most Oracle Parallel Server systems.

> **Additional Information:** See *Oracle8 Parallel Server Concepts and Administration* for more information about this background process.

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

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

These processes also execute job requests created by the DBMS_JOB package and propagate queued messages to queues on other databases (see "Oracle Advanced Queuing" on page 16-4).

> **Additional Information:** See *Oracle8 Administrator's Guide* for more information about this background process and job queues.

Queue Monitor Processes (QMN*n*) The *queue monitor process* is an optional background process for Oracle Advanced Queuing (Oracle AQ) which monitors the message queues. You can configure up to ten queue monitor processes. These processes, like the SNP*n* processes, are different from other Oracle background processes in that process failure does not cause the instance to fail.

See "Oracle Advanced Queuing" on page 16-4 for more information on message queues and the queue monitor process.

Dispatcher Processes (D*nnn*) The *dispatcher processes* support multithreaded configuration by allowing user processes to share a limited number of server processes. (See "The Multithreaded Server" on page 7-20.) With the multithreaded server, fewer shared server processes are required for the same number of users; therefore, the multithreaded server can support a greater number of users, particularly in cli-

ent/server environments where the client application and server operate on different machines.

You can create multiple dispatcher processes for a single database instance; at least one dispatcher must be created for each network protocol used with Oracle. The database administrator should start an optimal number of dispatcher processes depending on the operating system limitation on the number of connections per process, and can add and remove dispatcher processes while the instance runs.

> **Note:** Each user process that connects to a dispatcher must do so through Net8 or SQL*Net Version 2, even if both processes are running on the same machine.

In a multithreaded server configuration, a network listener process waits for connection requests from client applications, and routes each to a dispatcher process. If it cannot connect a client application to a dispatcher, the listener process starts a dedicated server process, and connects the client application to the dedicated server. The listener process is not part of an Oracle instance; rather, it is part of the networking processes that work with Oracle.

> **Additional Information:** See "The Multithreaded Server" on page 7-20 and the *Oracle Net8 Administrator's Guide* for more information about the network listener.

Shared Server Processes (S*nnn*) Each *shared server process* serves multiple client requests in the multithreaded server configuration. For more information, see "Shared Server Processes" on page 7-23.

Trace Files and the ALERT File

Each server and background process can write to an associated *trace file*. When a process detects an internal error, it dumps information about the error to its trace file. If an internal error occurs and information is written to a trace file, the administrator should contact Oracle support.

> **Additional Information:** See *Oracle8 Error Messages* for information about error 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 job queue processes (SNP*n*).

Additional information in trace files can provide guidance for tuning applications or an instance. Background processes always write this information to a trace file when appropriate. However, server processes write tuning information to a trace file only if the initialization parameter SQL_TRACE is set to TRUE for the instance or session. (Information about internal errors is always written to trace files.)

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 the SQL statements CREATE / ALTER / DROP DATABASE / TABLESPACE / ROLLBACK SEGMENT and the Oracle Enterprise Manager or Server Manager statements STARTUP, SHUTDOWN, ARCHIVE LOG, and RECOVER
- several messages and errors relating to the functions of shared server and dispatcher processes
- errors during the automatic refresh of a snapshot

Oracle uses the ALERT file to keep a record of these events as an alternative to displaying the information on an operator's console. (Many systems also display this information on the console.) If an administrative operation is successful, a message is written in the ALERT file as "completed" along with a timestamp.

Variations in Oracle Configuration

In a multiple-process Oracle instance, the code for connected users can be configured in one of three ways:

single-task Oracle	For each user, both the database application and the Oracle server code are combined in a single user process.
dedicated server (two-task Oracle)	For each user, the database application is run by a different process (a user process) than the one that executes the Oracle server code (a dedicated server process).
multithreaded server	The database application is run by a different process (a user process) than the one that executes the Oracle server code; each server process that executes Oracle server code (a *shared server process*) can serve multiple user processes.

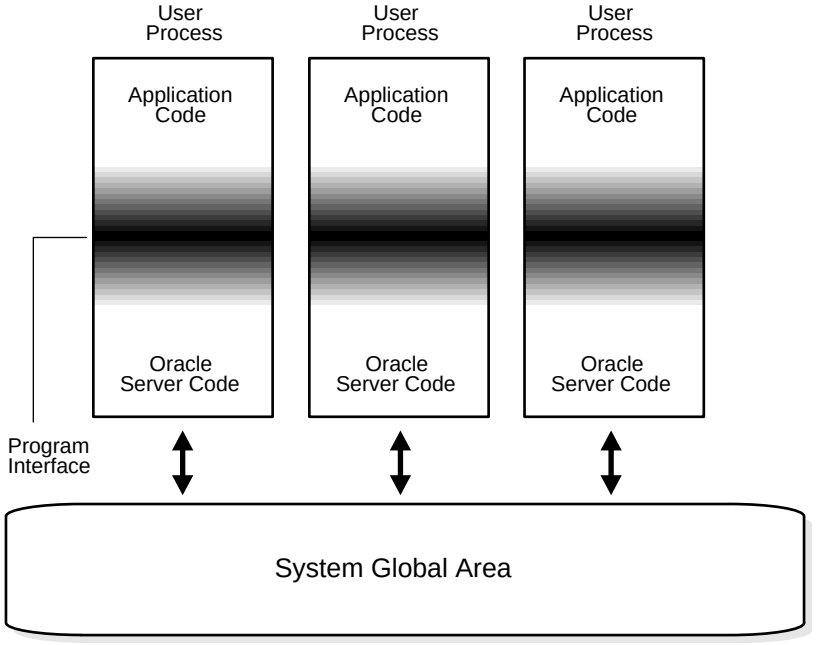
The following sections describe each variation in more detail.

> **Additional Information:** Some operating systems offer a choice of configurations; see your Oracle operating-system-specific documentation for more details on your options.

Single-Task Configuration

Figure 7–4 illustrates the single-task Oracle configuration. In this configuration, the database application and the Oracle server code all run in the same process, called a *user* process.

Figure 7–4 Oracle Using Combined User/Server Processes



This configuration of Oracle is feasible only 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 and server code.

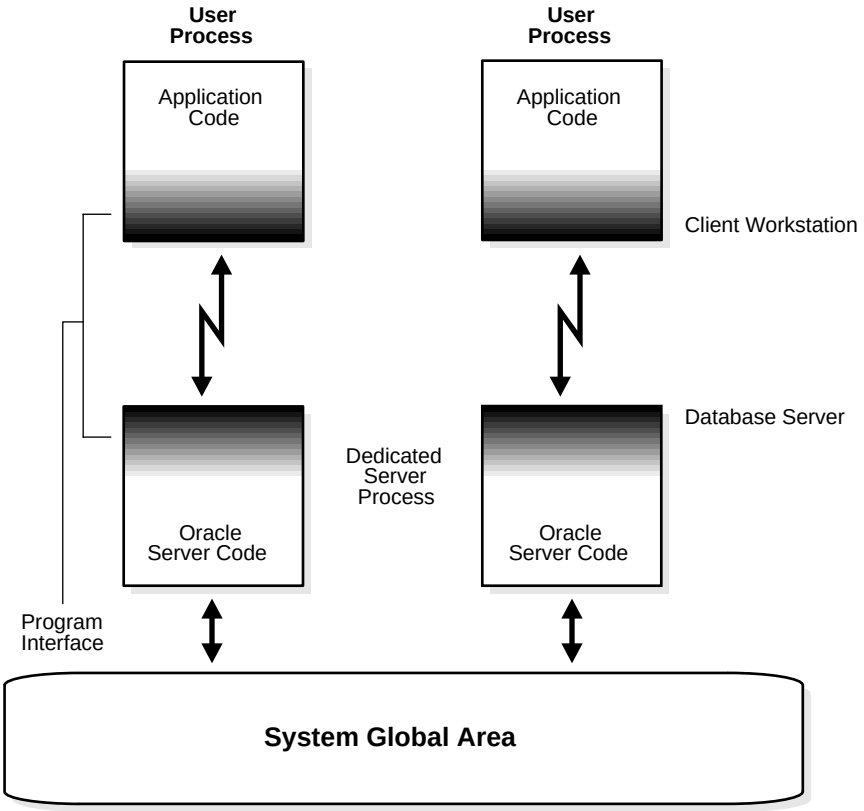
Note: The *program interface* is responsible for the separation and protection of the Oracle server code and is responsible for passing data between the database application and the Oracle user program. See "The Program Interface" on page 7-27.

Only one Oracle connection is allowed at any time by a process using the single-task configuration. However, in a user-written program it is possible to maintain this type of connection while concurrently connecting to Oracle using a network (Net8) interface.

Dedicated Server (Two-Task) Configuration

Figure 7–5 illustrates Oracle running on two computers using the dedicated server architecture. In this configuration, a user process executes the database application on one machine and a server process executes the associated Oracle server on another machine.

Figure 7–5 Oracle Using Dedicated Server Processes



The user and server processes are separate, distinct processes. The separate server process created on behalf of each user process is called a *dedicated server process* (or shadow process) because this server process acts only on behalf of the associated user process.

This configuration maintains a one-to-one ratio between the number of user processes and server processes. Even when the user is not actively making a database request, the dedicated server process remains (though it is inactive and may be paged out on some operating systems).

Figure 7–5 shows user and server processes running on separate computers connected across a network. However, the dedicated server architechture is also used if the same computer executes both the client application and the Oracle server code but the host operating system could not maintain the separation of the two programs if they were run in a single process. (UNIX is a common example of such an operating system.)

In the dedicated server configuration, the user and server processes communicate using different mechanisms:

- If the system is configured so that the user process and the dedicated server process run on the same computer, the program interface uses the host operating system's interprocess communication mechanism to perform its job.
- If the user process and the dedicated server process run on different computers, the program interface provides the communication mechanisms (such as the network software and Net8) between the programs.

  Additional Information: These communications links are operating system and installation dependent; see your Oracle operating-system-specific documentation and the Net8 documentation for more information.

Dedicated server architecture can sometimes result in inefficiency. Consider an order entry system with dedicated server processes. A customer places an order as a clerk enters the order into the database. For most of the transaction, the clerk is talking to the customer while the server process dedicated to the clerk's user process remains idle. The server process is not needed during most of the transaction, and the system is slower for other clerks entering orders. For applications such as this, the multithreaded server architecture may be preferable.

See "An Example of Oracle Using Dedicated Server Processes" on page 7-25 for a detailed illustration of the use of dedicated servers.

The Multithreaded Server

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

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

A number of different processes are needed in a multithreaded server system:

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

The multithreaded server requires Net8 or SQL*Net Version 2.

> **Note:** To use shared servers, a user process must connect through Net8 or SQL*Net Version 2, even if the process runs on the same machine as the Oracle instance.

When an instance starts, the network listener process opens and establishes a communication pathway through which users connect to Oracle. Then, each dispatcher process gives the listener process an address at which the dispatcher listens for connection requests. At least one dispatcher process must be configured and started for each network protocol that the database clients will use.

When a user process makes a connection request, the listener examines the request and determines whether the user process can use a shared server process. If so, the listener returns the address of the dispatcher process that has the lightest load and the user process connects to the dispatcher directly.

Some user processes cannot communicate with the dispatcher (such as those that connect using pre-Version 2 SQL*Net) so the network listener process cannot connect them to a dispatcher. In this case, or if the user process requests a dedicated server (see “Restricted Operations of the Multithreaded Server” on page 7-24), the listener creates a dedicated server and establishes an appropriate connection.

Additional Information: See the *Oracle Net8 Administrator's Guide* for more information about the network listener.

Dispatcher Request and Response Queues

A request from a user is a single program interface call that is part of the user's SQL statement. When a user makes a call, its dispatcher places the request on the *request queue,* where it is picked up by the next available shared server process.

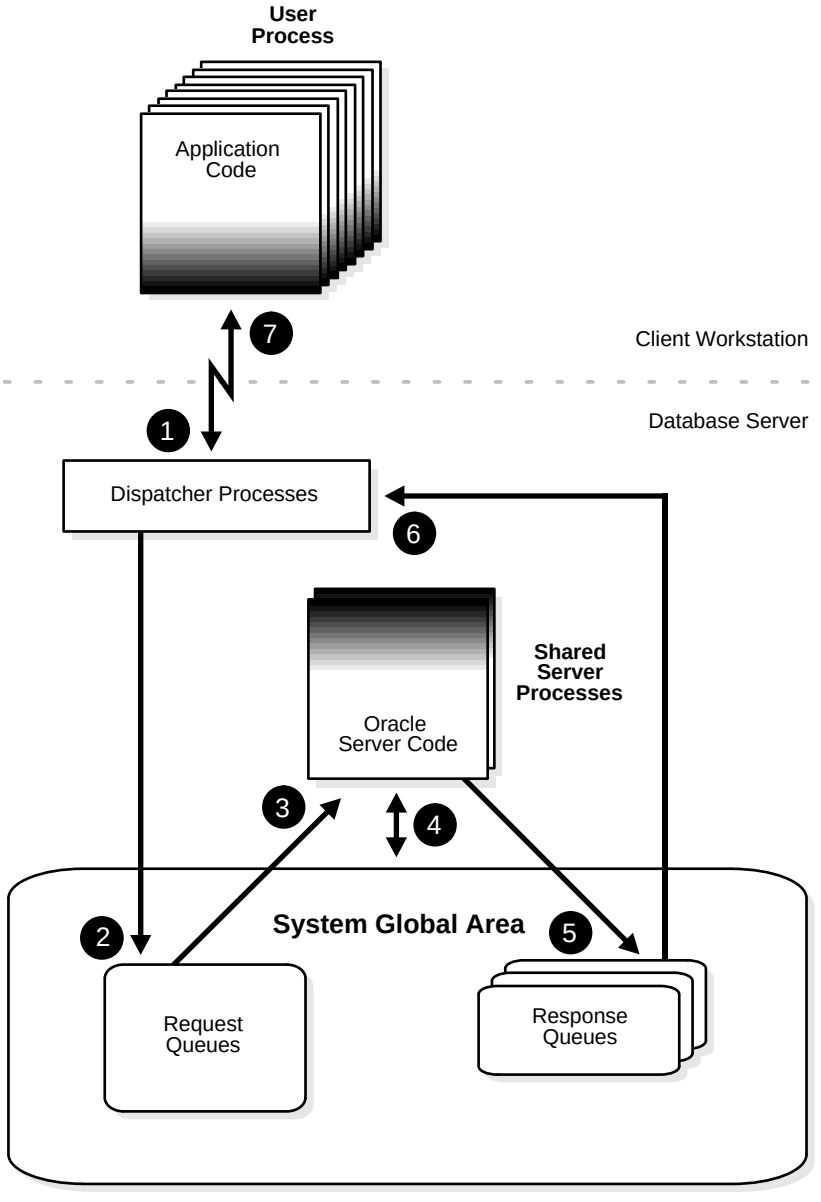
The request queue is in the SGA and is common to all dispatcher processes of an instance. The shared server processes check the common request queue for new requests, picking up new requests on a first-in-first-out basis. One shared server process picks up one request in the queue and makes all necessary calls to the database to complete that request.

When the server completes the request, it places the response on the calling dispatcher's *response queue*. Each dispatcher has its own response queue in the SGA. The dispatcher then returns the completed request to the appropriate user process.

For example, in an order entry system each clerk's user process connects to a dispatcher and each request made by the clerk is sent to that dispatcher, which places the request in the request queue. The next available shared server process picks up the request, services it, and puts the response in the response queue. When a clerk's request is completed, the clerk remains connected to the dispatcher but the shared server process that processed the request is released and available for other requests. While one clerk is talking to a customer, another clerk can use the same shared server process.

Figure 7–6 illustrates how user processes communicate with the dispatcher across the program interface and how the dispatcher communicates users' requests to shared server processes.

Figure 7–6 The Multithreaded Server Configuration and Shared Server Processes



Shared Server Processes

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

The PGA of a shared server process does not contain user-related data (which needs to be accessible to all shared server processes). The PGA of a shared server process contains only stack space and process-specific variables. “Program Global Areas (PGA)” on page 6-13 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 25, “Controlling 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 values of the initialization parameters MTS_SERVERS and MTS_MAX_SERVERS.

Artificial Deadlocks

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

1. One user acquires an exclusive lock on a resource by issuing a SELECT statement with the FOR UPDATE clause or a LOCK TABLE statement.
2. The shared server process that processes the locking request is released once the statement completes.
3. Other users attempt to access the locked resource. Each shared server process is bound to the user process it is serving until the necessary locked resource becomes available. Eventually, all shared servers may be bound to user processes waiting for locked resources.
4. The original user attempts to submit a new request (such as a COMMIT or ROLLBACK statement) to release the previously acquired lock, but cannot because all shared server processes are currently being used.

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

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

Restricted Operations of the Multithreaded Server

Certain administrative activities cannot be performed while connected to a dispatcher process, including shutting down or starting an instance and media recovery. An error message is issued if you attempt to perform these activities while connected to a dispatcher process.

These activities are typically performed when connected with administrator privileges. When you want to connect with administrator privileges in a system configured with multithreaded servers, you must state in your connect string that you want to use a dedicated server process (SRVR=DEDICATED) instead of a dispatcher process.

> **Additional Information:** See your Oracle operating system-specific documentation or the *Oracle Net8 Administrator's Guide* for the proper connect string syntax.

See "An Example of Oracle Using the Multithreaded Server" on page 7-26 for a detailed illustration of the use of the multithreaded server configuration.

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 multithreaded server Oracle configurations.

An Example of Oracle Using Dedicated Server Processes

The following example is a simple illustration of the dedicated server architecture. These steps show only the most basic level of operations that Oracle performs.

1. A database server machine is currently running Oracle using multiple background processes.
2. A user process on a client workstation runs a database application such as SQL*Plus. The client application attempts to establish a connection to the server using a Net8 driver.
3. The database server is currently running the proper Net8 driver. The network listener process on the database server detects the connection request from the client database application and creates a dedicated server process on the database server on behalf of the user process.
4. The user executes a single SQL statement. For example, the user inserts a row into a table.
5. The dedicated server process receives the statement. At this point, two paths can be followed to continue processing the SQL statement:
   - If the shared pool contains a shared SQL area for an identical SQL statement, the server process uses the existing shared SQL area to execute the client's SQL statement.
   - If the shared pool does not contain a shared SQL area for 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 DBW0 process determines it is most efficient to do so. The LGWR process records the transaction in the online redo log file only on a subsequent commit request from the user.
8. If the request is successful, the server sends a message across the network to the user. If it is not successful, an appropriate error message is transmitted.
9. Throughout this entire procedure, the other background processes are running and watching for any conditions that require intervention. In addition, Oracle

is managing other transactions and preventing contention between different transactions that request the same data.

An Example of Oracle Using the Multithreaded Server

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

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

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

4. The user issues a single SQL statement. For example, 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. (In Step 5, 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:

- provides a security barrier, preventing destructive access to the SGA by client user processes
- acts as a communication mechanism, formatting information requests, passing data, and trapping and returning errors
- converts and translates data, particularly between different types of computers or to external user program datatypes

The *Oracle code* acts as a server, performing database tasks on behalf of an *application* (a client), such as fetching rows from data blocks. It consists of several parts, provided by both Oracle software and operating-system-specific software.

Program Interface Structure

The program interface consists of the following pieces:

- Oracle call interface (OCI) or the Oracle runtime library (SQLLIB)
- the client or user side of the program interface (also called the *UPI*)
- various *Net8 drivers* (protocol-specific communications software)
- operating system communications software
- the server or Oracle side of the program interface (also called the *OPI*)

Both the user and Oracle sides of the program interface execute Oracle software, as do the drivers.

Net8 is the portion of the program interface that allows the client application program and the Oracle server to reside on separate computers in your communication network.

The Program Interface Drivers

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

You may install multiple drivers (such as the asynchronous or DECnet drivers), and select one as the default driver, but allow an individual user to use other drivers by specifying the desired driver at the time of connection. Different processes

can use different drivers. A single process can have concurrent connections to a single database or to multiple databases (either local or remote) using different Net8 drivers.

The installation and configuration guide and Net8 documentation for your system contains details about choosing and installing drivers and adding new drivers after installation. The Net8 documentation describes selecting a driver at runtime while accessing Oracle.

> **Additional Information:** See *Oracle Net8 Administrator's Guide* for more information about Net8.

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.

> **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.

Part IV

The Object-Relational DBMS

Part IV describes the Oracle relational model for database management and the object extensions to that model.

Part IV contains the following chapters:

- Chapter 8, “Schema Objects”
- Chapter 9, “Partitioned Tables and Indexes”
- Chapter 10, “Built-In Datatypes”
- Chapter 11, “User-Defined Datatypes (Objects Option)”
- Chapter 12, “Using User-Defined Datatypes”
- Chapter 13, “Object Views”

8

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 database objects contained in a user's schema. It includes:

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

> **Additional Information:** If you are using Trusted Oracle, see your *Trusted Oracle* documentation for information about schema objects in that environment.

Overview of Schema Objects

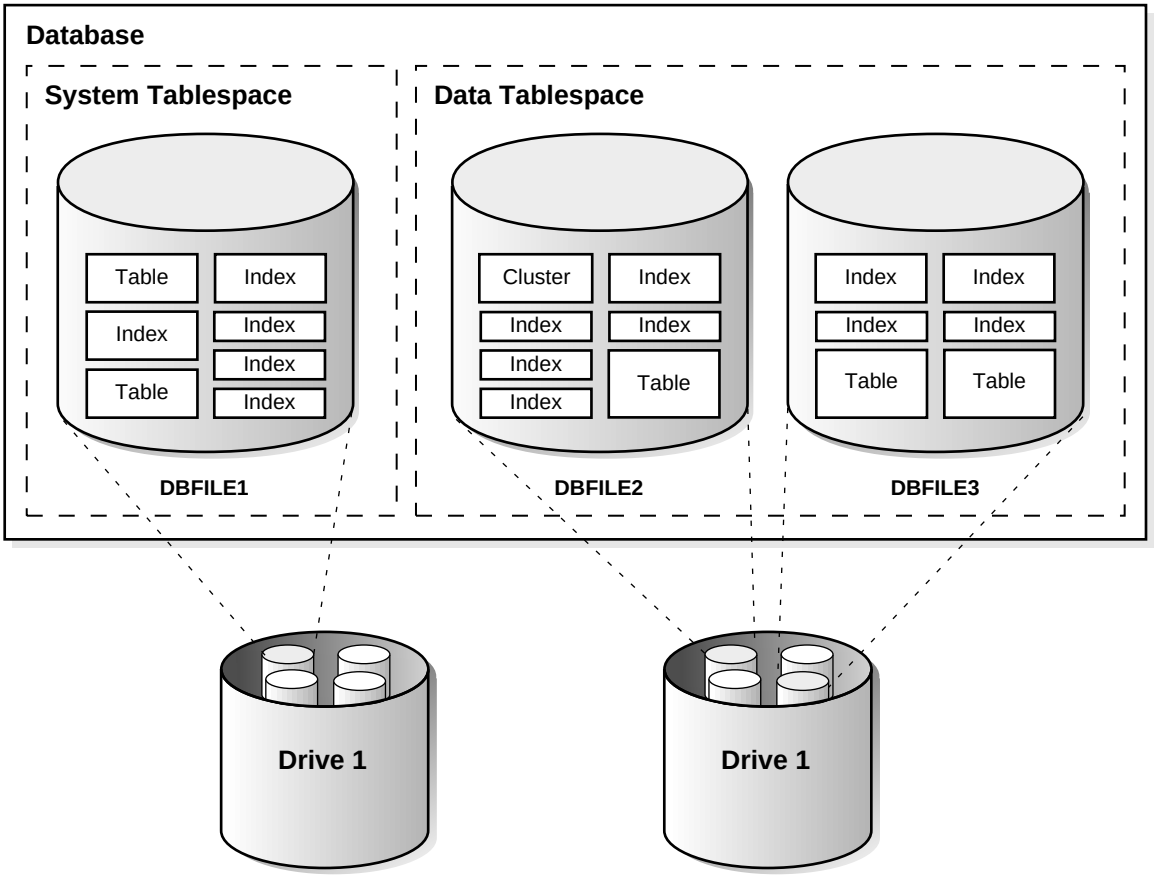
Associated with each database user is a *schema*. A schema is a collection of schema objects. Examples of schema objects include tables, views, sequences, synonyms, indexes, clusters, database links, snapshots, procedures, functions, and packages.

Schema objects are logical data storage structures. Schema objects do not have a one-to-one correspondence to physical files on disk that store their information. However, Oracle stores a schema object logically within a tablespace of the database. The data of each object is physically contained in one or more of the tablespace's datafiles. For some objects such as tables, indexes, and clusters, you can specify how much disk space Oracle allocates for the object within the tablespace's datafiles.

There is no relationship between schemas and tablespaces: a tablespace can contain objects from different schemas, and the objects for a schema can be contained in different tablespaces. Figure 8–1 illustrates the relationship among objects, tablespaces, and datafiles.

> **Additional Information:** This chapter explains tables, views, sequences, synonyms, indexes, and clusters. Other kinds of schema objects are explained elsewhere in this manual, or in other manuals. Specifically:
>
> - Database links are described in Chapter 30, "Distributed Databases".
> - Object views are described in Chapter 13, "Object Views" and in the *Oracle8 Application Developer's Guide*.
> - Procedures, functions, and packages are described in Chapter 17, "Procedures and Packages".
> - Triggers are described in Chapter 18, "Database Triggers".
> - Snapshots are described in *Oracle8 Replication*.

Figure 8–1 Schema Objects, Tablespaces, and Datafiles



Tables

Tables are the basic unit of data storage in an Oracle database. Data is stored in *rows* and *columns*. You define a table with a *table name* (such as EMP) and set of columns. You give each column a *column name* (such as EMPNO, ENAME, and JOB), a *datatype* (such as VARCHAR2, DATE, or NUMBER), and a *width* (the width might be predetermined by the datatype, as in DATE) or *precision* and *scale* (for columns of the NUMBER datatype only). A row is a collection of column information corresponding to a single record.

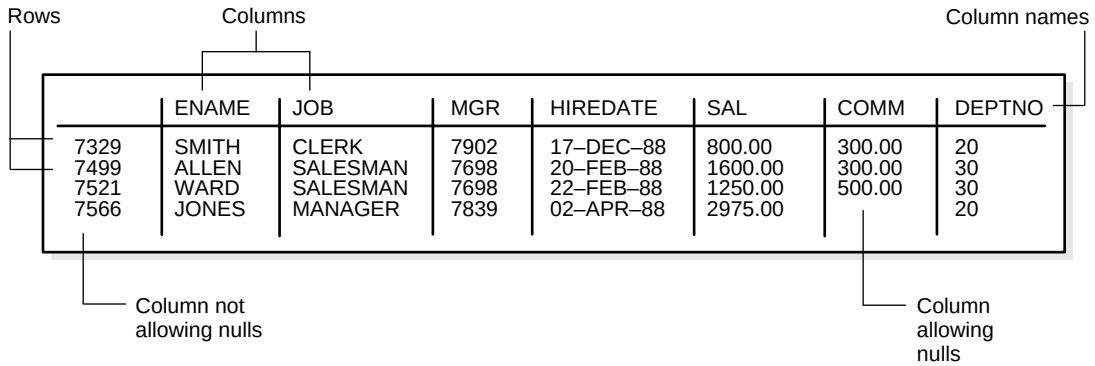
See Chapter 10, “Built-In Datatypes”, for a discussion of the Oracle datatypes.

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 24, "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 8–2 shows a sample table named EMP.

Figure 8–2 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 25-8.

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 and Migrating" on page 2-10.

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.

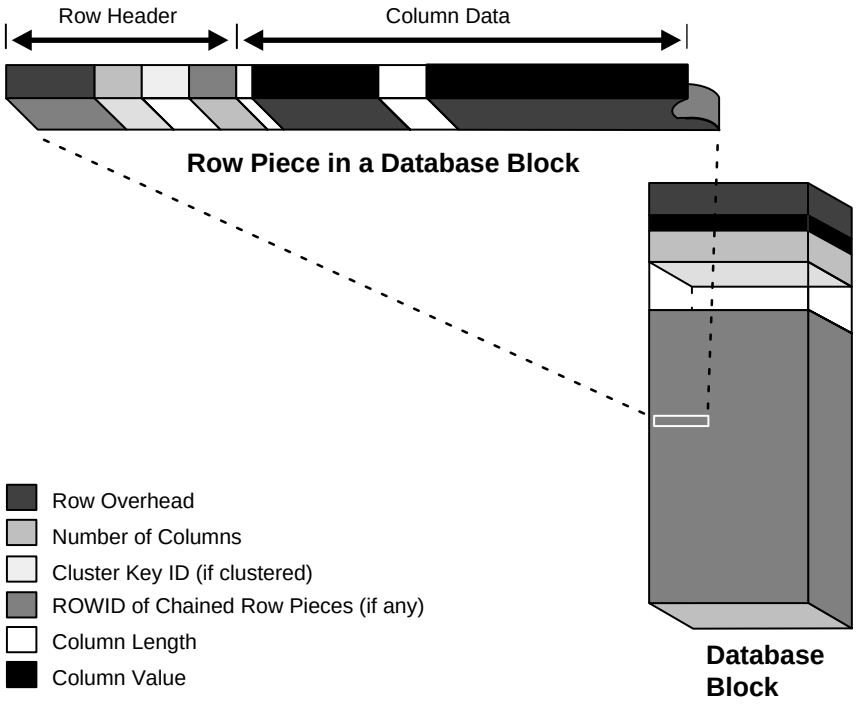
Figure 8–3 shows the format of a row piece.

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.

Figure 8–3 The Format of a Row Piece



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 8-32.

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 “ROWID Datatype” on page 10-12 for more information.

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.

> **Additional Information:** See *Oracle8 SQL Reference* for more information about comparisons using IS NULL and the NVL function.

Nulls are not indexed, except when the cluster key column value is null or the index is a bitmap index (see “Bitmap Indexes and Nulls” on page 8-27).

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 SQL functions SYSDATE, USER, USERENV, and UID. The datatype of the default literal or expression must match or be convertible to the column datatype.

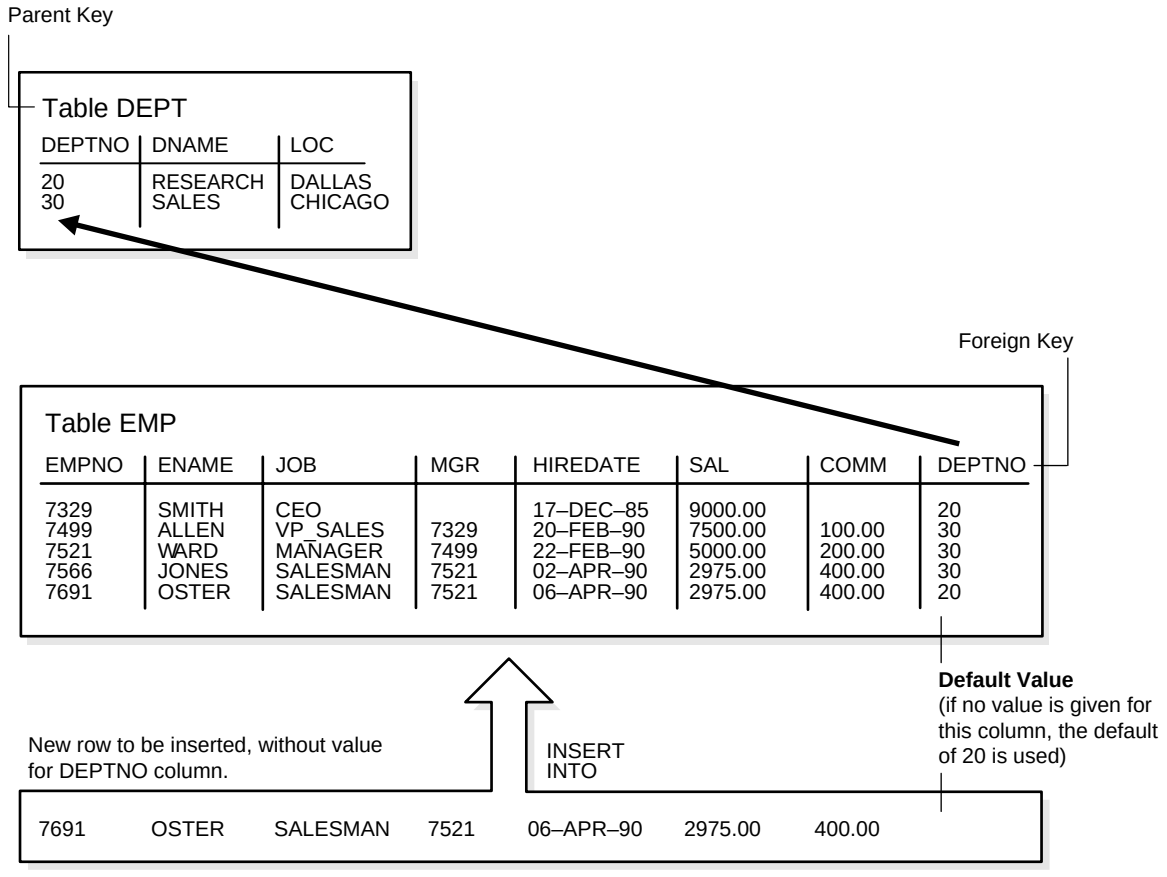
If a default value is not explicitly defined for a column, the default for the column is implicitly set to NULL.

Default Value Insertion and Integrity Constraint Checking

Integrity constraint checking occurs after the row with a default value is inserted. For example, in Figure 8–4, a row is inserted into the EMP table that does not include a value for the employee’s department number. Because no value is supplied for the department number, Oracle inserts the DEPTNO column’s default value “20”. After inserting the default value, Oracle checks the FOREIGN KEY integrity constraint defined on the DEPTNO column.

For more information about integrity constraints, see Chapter 24, “Data Integrity”.

Figure 8–4 DEFAULT Column Values



Nested Tables

In the Oracle object-relational database, you can create a table with a column whose datatype is another table. That is, tables can be *nested* within other tables as values in a column. The Oracle server stores nested table data "out of line" from the rows of the parent table, using a *store table* which is associated with the nested table column. The parent row contains a unique set identifier value associated with a nested table instance.

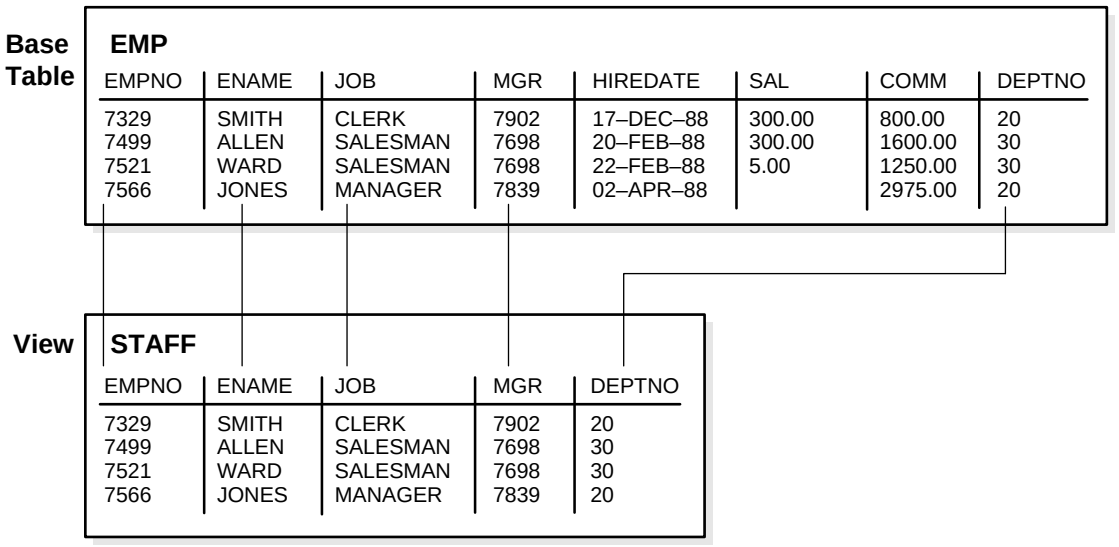
Additional Information: See *Oracle8 Application Developer's Guide.*

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 8–5 shows an example of a view called STAFF derived from the base table EMP. Notice that the view shows only five of the columns in the base table.

Figure 8–5 An Example of a View



Since views are derived from tables, they have many similarities. For example, you can define views with up to 1000 columns, just like a table. You can query views, and with some restrictions you can update, insert into, and delete from views. All operations performed on a view actually affect data in some base table of the view and are subject to the integrity constraints and triggers of the base tables.

Additional Information: See *Oracle8 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 8–5 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.

  > **Additional Information:** See the *Oracle8 SQL Reference* for information about GROUP BY or UNION.

- 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.

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 recreate it, Oracle determines whether the new base table is acceptable to the existing definition of the view. See Chapter 19, "Oracle Dependency Management", 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 8–1 lists rules for updatable join views.

Table 8–1 Rules for INSERT, UPDATE, and DELETE on Join Views

Rule	Description
General Rule	Any INSERT, UPDATE, or DELETE operation on a join view can modify only one underlying base table at a time.
UPDATE Rule	All updatable columns of a join view must map to columns of a key preserved table. If the view is defined with the WITH CHECK OPTION clause, then all join columns and all columns of repeated tables are 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, INSERT statements are not permitted.

Views that are not updatable can be modified using INSTEAD OF triggers. See "INSTEAD OF Triggers" on page 18-11 for more information.

Object Views

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

> **Additional Information:** See Chapter 13, "Object Views" and the *Oracle8 Application Developer's Guide*.

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.

> **Additional Information:** For performance implications when using sequences, see the *Oracle8 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 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:

- A table called SALES_DATA is in the schema owned by the user JWARD.
- The SELECT privilege for the SALES_DATA table is granted to PUBLIC.

At this point, you 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.

Oracle provides several indexing schemes, which provide complementary performance functionality: B*-tree indexes (currently the most common), B*-tree cluster indexes, hash cluster indexes, reverse key indexes, and bitmap indexes.

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 affecting the base tables or other indexes. If you drop an index, all applications continue to work; however, access of previously indexed data might be slower. Indexes, as independent structures, require storage space.

Oracle automatically maintains and uses indexes 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.

The optimizer can use an existing index to build another index. This results in a much faster index build.

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 col-

umns. Oracle enforces UNIQUE integrity constraints by automatically defining a unique index on the unique key.

Composite Indexes

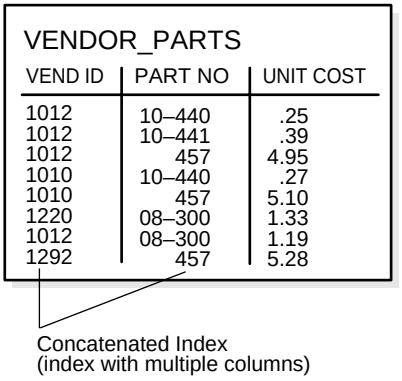
A *composite index* (also called a *concatenated index*) is an index that you create on multiple columns in a table. Columns in a composite index can appear in any order and need not be adjacent in the table.

Composite indexes can speed retrieval of data for SELECT statements in which the WHERE clause references all or the leading portion of the columns in the composite index. Therefore, the order of the columns used in the definition is important; generally, the most commonly accessed or most selective columns go first.

Additional Information: See *Oracle8 Tuning* for more information.

Figure 8–6 illustrates the VENDOR_PARTS table that has a composite index on the VENDOR_ID and PART_NO columns.

Figure 8–6 Indexes, Primary Keys, Unique Keys, and Foreign Keys



No more than 32 columns can form a regular composite index, and for a bitmap index the maximum number columns is 30. A key value cannot exceed roughly one-half (minus some overhead) the available data space in a data block.

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 24, “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 25-8.

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.

> **Additional Information:** See your Oracle operating-system-specific documentation for information about the overhead of an index block.

When you create an index, Oracle fetches and sorts the columns to be indexed, and stores the ROWID along with the index value for each row. Then Oracle loads the index from the bottom up. For example, consider the statement:

```
CREATE INDEX emp_ename ON emp(ename);
```

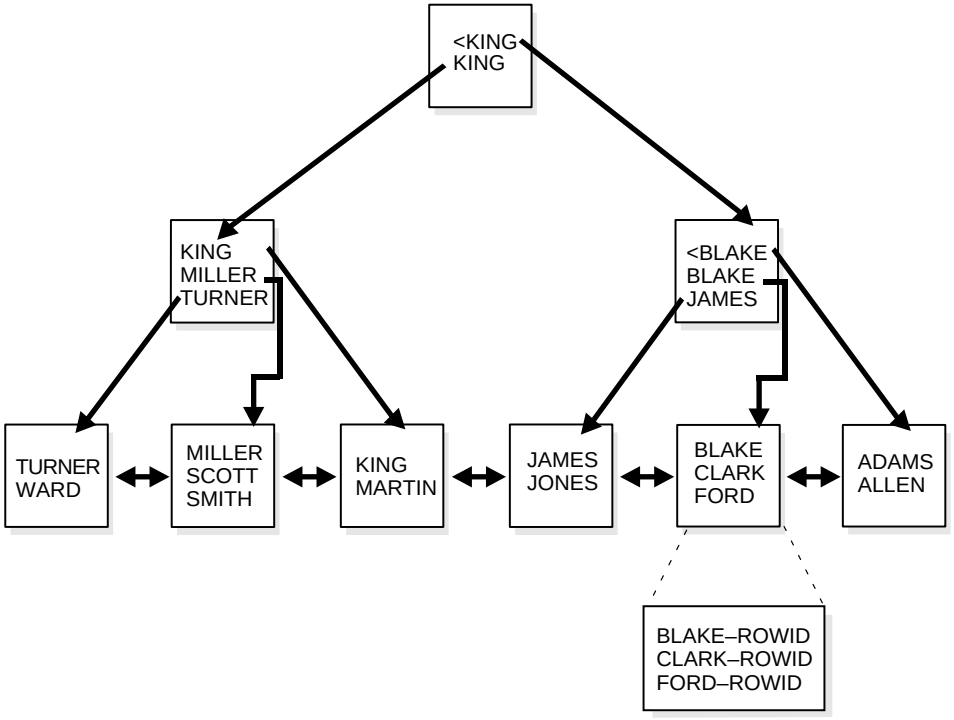
Oracle sorts the EMP table on the ENAME column. It then loads the index with the ENAME and corresponding ROWID values in this sorted order. When it uses the index, Oracle does a quick search through the sorted ENAME values and then uses the associated ROWID values to locate the rows having the sought ENAME value.

Although Oracle accepts the keywords ASC, DESC, COMPRESS, and NOCOMPRESS in the CREATE INDEX 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 8–7 illustrates the structure of a B*-tree index.

Figure 8–7 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.

Advantages of B*-Tree Structure

The B*-tree structure has the following advantages:

- All leaf blocks of the tree are at the same depth, so retrieval of any record from anywhere in the index takes approximately the same amount of time.
- B*-tree indexes automatically stay balanced.
- All blocks of the B*-tree are three-quarters full on the average.
- B*-trees provide excellent retrieval performance for a wide range of queries, including exact match and range searches.
- Inserts, updates, and deletes are efficient, maintaining key order for fast retrieval.
- B*-tree performance is good for both small and large tables, and does not degrade as the size of a table grows.

Reverse Key Indexes

Creating a *reverse key index*, compared to a standard index, reverses the bytes of each column indexed (except the ROWID) while keeping the column order. Such an arrangement can help avoid performance degradation in indexes in an Oracle Parallel Server environment where modifications to the index are concentrated on a small set of leaf blocks. By reversing the keys of the index, the insertions become distributed across all leaf keys in the index.

Using the reverse key arrangement eliminates the ability to run an index range scanning query on the index. Because lexically adjacent keys are not stored next to each other in a reverse-key index, only fetch-by-key or full-index (table) scans can be performed.

Under some circumstances using a reverse-key index can make an OLTP Oracle Parallel Server application faster. For example, keeping the index of mail messages in Oracle Office: some users keep old messages around, and the index must maintain pointers to these as well as to the most recent.

The REVERSE keyword provides a simple mechanism for creating a reverse key index. You can specify the keyword REVERSE along with the optional index specifications in a CREATE INDEX statement:

```
CREATE INDEX i ON t (a,b,c) REVERSE;
```

You can specify the keyword NOREVERSE to REBUILD a reverse-key index into one that is not reverse keyed:

```
ALTER INDEX i REBUILD NOREVERSE;
```

Rebuilding a reverse-key index without the NOREVERSE keyword produces a rebuilt, reverse-key index. You cannot rebuild a normal index as a reverse key index; you must use the CREATE command instead.

Bitmap Indexes

> **Attention:** Bitmap indexes are available only if you have purchased the Oracle8 Enterprise Edition. See *Getting to Know Oracle8 and the Oracle8 Enterprise Edition* for more information about the features available in Oracle8 and the Oracle8 Enterprise Edition.

The purpose of an index is to provide pointers to the rows in a table that contain a given key value. In a regular index, this is achieved by storing a list of ROWIDs for each key corresponding to the rows with that key value. (Oracle stores each key value repeatedly with each stored ROWID.) In a *bitmap index*, a bitmap for each key value is used instead of a list of ROWIDs.

Each bit in the bitmap corresponds to a possible ROWID, and if the bit is set, it means that the row with the corresponding ROWID contains the key value. A mapping function converts the bit position to an actual ROWID, so the bitmap index provides the same functionality as a regular index even though it uses a different representation internally. If the number of different key values is small, bitmap indexes are very space efficient.

Bitmap indexing efficiently merges indexes that correspond to several conditions in a WHERE clause. Rows that satisfy some, but not all conditions are filtered out before the table itself is accessed. This improves response time, often dramatically.

Benefits for Data Warehousing Applications

Bitmap indexing benefits data warehousing applications which have large amounts of data and ad hoc queries, but a low level of concurrent transactions. For such applications, bitmap indexing provides:

- reduced response time for large classes of ad hoc queries
- a substantial reduction of space usage compared to other indexing techniques
- dramatic performance gains even on very low end hardware
- very efficient parallel DML and loads

Fully indexing a large table with a traditional B*-tree index can be prohibitively expensive in terms of space since the index can be several times larger than the data in the table. Bitmap indexes are typically only a fraction of the size of the indexed data in the table.

Bitmap indexes are not suitable for OLTP applications with large numbers of concurrent transactions modifying the data. These indexes are primarily intended for decision support (DSS) in data warehousing applications where users typically query the data rather than update it.

Bitmap indexes are integrated with the Oracle cost-based optimization approach and execution engine. They can be used seamlessly in combination with other Oracle execution methods. For example, the optimizer can decide to perform a hash join between two tables using a bitmap index on one table and a regular B*-tree index on the other. The optimizer considers bitmap indexes and other available access methods, such as regular B*-tree indexes and full table scan, and chooses the most efficient method, taking parallelism into account where appropriate.

Parallel query and parallel DML work with bitmap indexes as with traditional indexes. (Bitmap indexes on partitioned tables must be local indexes; see "Index Partitioning" on page 9-22 for more information.) Parallel create index and concatenated indexes are also supported.

Cardinality

The advantages of using bitmap indexes are greatest for low cardinality columns: that is, columns in which the number of distinct values is small compared to the number of rows in the table. If the values in a column are repeated more than a hundred times, the column is a candidate for a bitmap index. Even columns with a lower number of repetitions (and thus higher cardinality), can be candidates if they tend to be involved in complex conditions in the WHERE clauses of queries.

For example, on a table with one million rows, a column with 10,000 distinct values is a candidate for a bitmap index. A bitmap index on this column can out-perform a B*-tree index, particularly when this column is often queried in conjunction with other columns.

B*-tree indexes are most effective for high-cardinality data: that is, data with many possible values, such as CUSTOMER_NAME or PHONE_NUMBER. A regular B*-tree index can be several times larger than the indexed data. Used appropriately, bitmap indexes can be significantly smaller than a corresponding B*-tree index.

In ad hoc queries and similar situations, bitmap indexes can dramatically improve query performance. AND and OR conditions in the WHERE clause of a query can be quickly resolved by performing the corresponding boolean operations directly

on the bitmaps before converting the resulting bitmap to ROWIDs. If the resulting number of rows is small, the query can be answered very quickly without resorting to a full table scan of the table.

Bitmap Index Example

Table 8–2 shows a portion of a company's customer data.

Table 8–2 Bitmap Index Example

CUSTOMER #	MARITAL_ STATUS	REGION	GENDER	INCOME_ LEVEL
101	single	east	male	bracket_1
102	married	central	female	bracket_4
103	married	west	female	bracket_2
104	divorced	west	male	bracket_4
105	single	central	female	bracket_2
106	married	central	female	bracket_3

Since MARITAL_STATUS, REGION, GENDER, and INCOME_LEVEL are all low-cardinality columns (there are only three possible values for marital status and region, two possible values for gender, and four for income level) it is appropriate to create bitmap indexes on these columns. A bitmap index should not be created on CUSTOMER# because this is a high-cardinality column. Instead, a unique B*-tree index on this column in order would provide the most efficient representation and retrieval.

Table 8–3 illustrates the bitmap index for the REGION column in this example. It consists of three separate bitmaps, one for each region.

Table 8–3 Sample Bitmap

REGION='east'	REGION='central'	REGION='west'
1	0	0
0	1	0
0	0	1
0	0	1
0	1	0
0	1	0

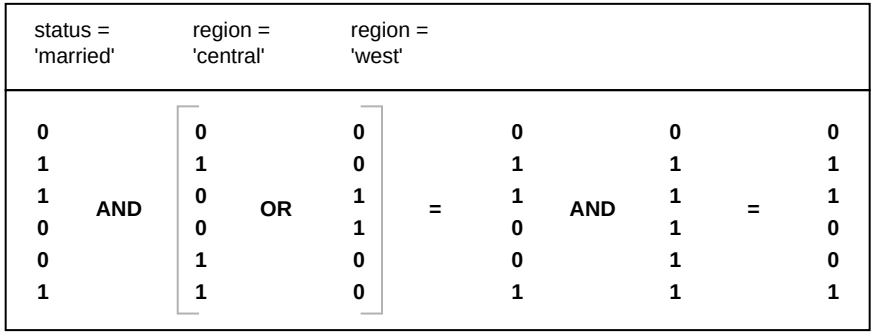
Each entry (or "bit") in the bitmap corresponds to a single row of the CUSTOMER table. The value of each bit depends upon the values of the corresponding row in the table. For instance, the bitmap REGION='east' contains a one as its first bit: this is because the region is "east" in the first row of the CUSTOMER table. The bitmap REGION='east' has a zero for its other bits because none of the other rows of the table contain "east" as their value for REGION.

An analyst investigating demographic trends of the company's customers might ask, "How many of our married customers live in the central or west regions?" This corresponds to the following SQL query:

```
SELECT COUNT(*) FROM CUSTOMER
    WHERE MARITAL_STATUS = 'married' AND REGION IN ('central','west');
```

Bitmap indexes can process this query with great efficiency by merely counting the number of ones in the resulting bitmap, as illustrated in Figure 8–8. To identify the specific customers who satisfy the criteria, the resulting bitmap would be used to access the table.

Figure 8–8 Executing a Query Using Bitmap Indexes



Bitmap Indexes and Nulls

Bitmap indexes include rows that have NULL values, unlike most other types of indexes. Indexing of nulls can be useful for some types of SQL statements, such as queries with the group function COUNT.

Example 1:

```
SELECT COUNT(*) FROM EMP;
```

Any bitmap index can be used for this query because all table rows are indexed, including those that have NULL data. If nulls were not indexed, the optimizer would only be able to use indexes on columns with NOT NULL constraints.

Example 2:

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

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

Example 3:

```
SELECT COUNT(*) FROM CUSTOMER
    WHERE GENDER = 'M' AND STATE != 'CA';
```

This query can be answered by finding the bitmap for GENDER = 'M' and substracting the bitmap for STATE = 'CA'. If STATE may contain null values (that is, if

it does not have a NOT NULL constraint), then the bitmaps for STATE = 'NULL' must also be subtracted from the result.

Bitmap Indexes on Partitioned Tables

Like other indexes, you can create bitmap indexes on partitioned tables. The only restriction is that bitmap indexes must be local to the partitioned table — they cannot be global indexes. (Global bitmap indexes are supported only on nonpartitioned tables).

For information about partitioned tables and descriptions of local and global indexes, see Chapter 9, "Partitioned Tables and Indexes".

> **Additional Information:** *Oracle8 Tuning* contains information about using bitmap indexes.

Index-Organized Tables

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

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

Index-organized tables are suitable for accessing data via primary key or via any key that is a valid prefix of the primary key. There is no duplication of key values because only non-key column values are stored with the key.

Applications manipulate the index-organized table just like a regular table, using SQL statements. However, the database system performs all operations by manipulating the corresponding B*-tree index.

Table 8–4 summarizes the differences between index-organized tables and regular tables.

Table 8–4 Comparison of Index-Organized Tables with Regular Tables

Regular Table	Index-Organized Table
ROWID uniquely identifies a row; primary key can be optionally specified	Primary key uniquely identifies a row; primary key must be specified
Implicit ROWID column; allows building physical secondary indexes	No implicit ROWID column; cannot have physical secondary indexes
ROWID based access	Primary key based access
Sequential scan returns all rows	Full-index scan returns all rows in primary key order
UNIQUE constraint and triggers allowed	UNIQUE constraint not allowed but triggers are allowed
A table can be stored in a cluster containing other tables.	An index-organized table cannot be stored in a cluster.
Distribution, replication, and partitioning supported	Distribution, replication, and partitioning not supported

Additional Information: See *Oracle8 Administrator's Guide* for information about how to create and maintain index-organized tables.

Benefits of Index-Organized Tables

Because rows are stored in the index, index-organized tables provide a faster key-based access to table data for queries involving exact match and/or range search. The storage requirements are reduced as key columns are not duplicated in the table and the index. The data row stored with the key only contains non-key column values. Also, placing the data row with the key eliminates the additional storage required for ROWIDs that link key values to corresponding rows in the case of an index for a regular table.

Index-Organized Tables with Row Overflow Area

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

Oracle provides a Row Overflow Area clause to overcome this problem. You can specify an overflow tablespace as well as a threshold value. The threshold is specified as percentage of the block size.

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

Applications of Interest for Index-Organized Tables

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

- Information Retrieval (IR) applications
- Spatial applications
- OLAP applications

Information Retrieval Applications

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

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

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

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

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

- Since the data row is stored along with the key, retrieval of occurrence data for an inverted index involves traversing the index and getting the data rows from the appropriate leaf nodes.
- Only the non-key column values are stored with the key in the index. Thus, there is no duplication of data. Also, this avoids the additional ROWID storage overhead which is required if an index is maintained on a regular table.

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

Spatial Applications

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

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

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

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

OLAP Applications

On-line analytical processing (OLAP) applications typically manipulate multi-dimensional blocks. To allow fast retrieval of portions of the multi-dimensional

blocks, they maintain an inverted index to map a set of dimension values to set of pages. An entry in the inverted index is of the form:

```
<dimension_value, list_of_pages>
```

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

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.

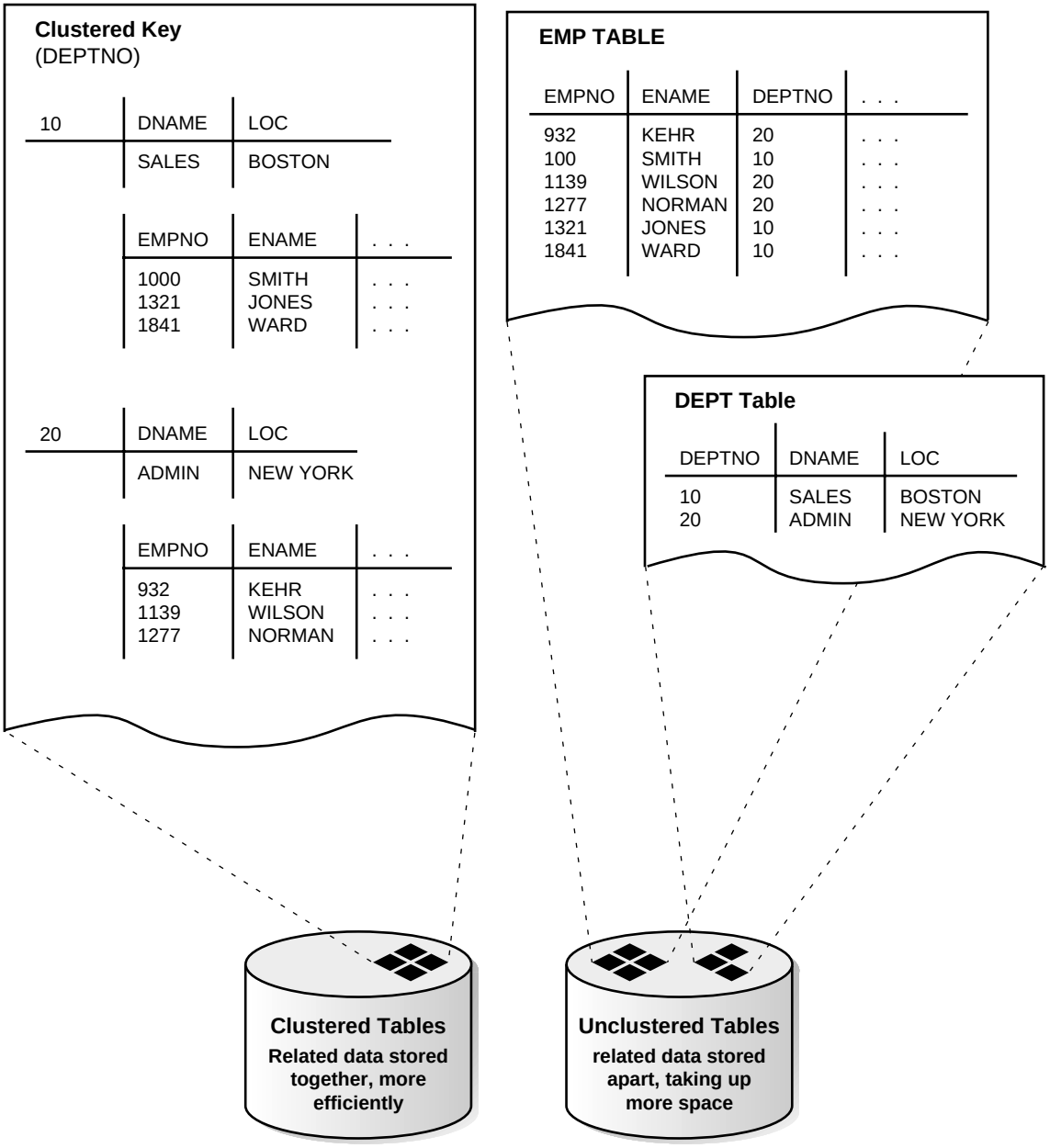
For example, the EMP and DEPT table share the DEPTNO column. When you cluster the EMP and DEPT tables (Figure 8–9, “Clustered Table Data”), Oracle physically stores all rows for each department from both the EMP and DEPT tables in the same data blocks.

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, in Figure 8–9 notice how each cluster key (each DEPTNO) is stored just once for many rows that contain the same value in both the EMP and DEPT tables.

Figure 8–9 Clustered Table Data



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.

> **Additional Information:** For more information about the performance implications of using clusters, see *Oracle8 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 recreate the cluster's index to allow access to data in the cluster.

Hash Clusters

Hashing is an optional way of storing table data to improve the performance of data retrieval. To use hashing, 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

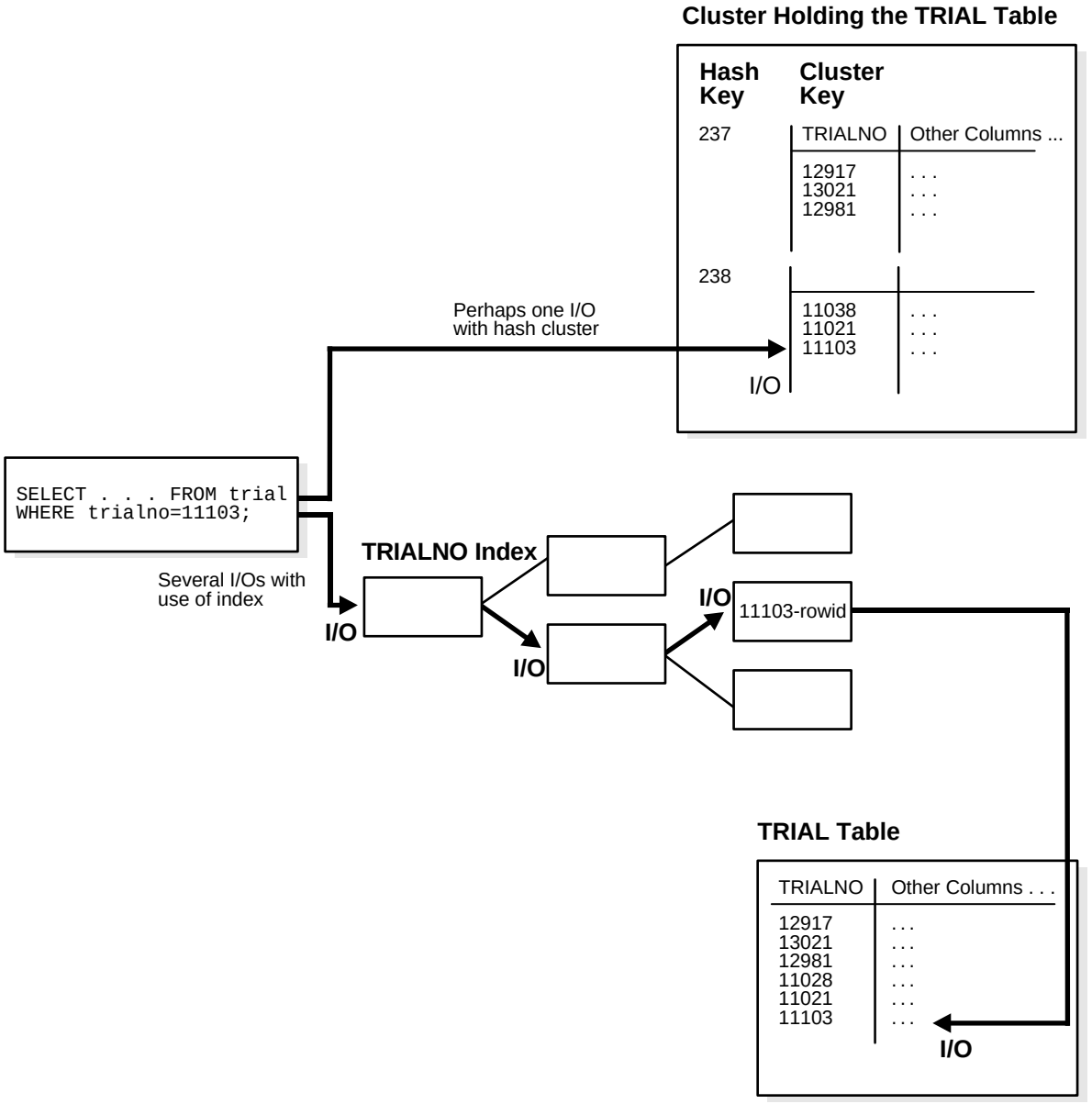
A hash cluster stores related rows together in the same data blocks. Rows in a hash cluster are stored together based on their hash value.

> **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 8–10 illustrates data retrieval for a table in a hash cluster as well as a table with an index. The following sections further explain the internal operations of hash cluster storage.

Figure 8–10 Hashing vs. Indexing: Data Storage and Information



Hash Key Values

To find or store a row in a hash cluster, Oracle applies the hash function to the row's cluster key value. The resulting hash value corresponds to a data block in the cluster, which Oracle then reads or writes on behalf of an issued statement. The number of hash values for a hash cluster is fixed at creation and is determined by the HASHKEYS parameter of the CREATE CLUSTER 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 consist of columns of any datatype (except LONG and LONG RAW). The internal hash function offers sufficient distribution of cluster key values among available hash keys, producing a minimum number of collisions for any type of cluster key.

Specifying the Cluster Key as the Hash Function

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

Instead of using the internal hash function to generate a hash value, Oracle checks the cluster key value. If the cluster key value is less than HASHKEYS, 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.

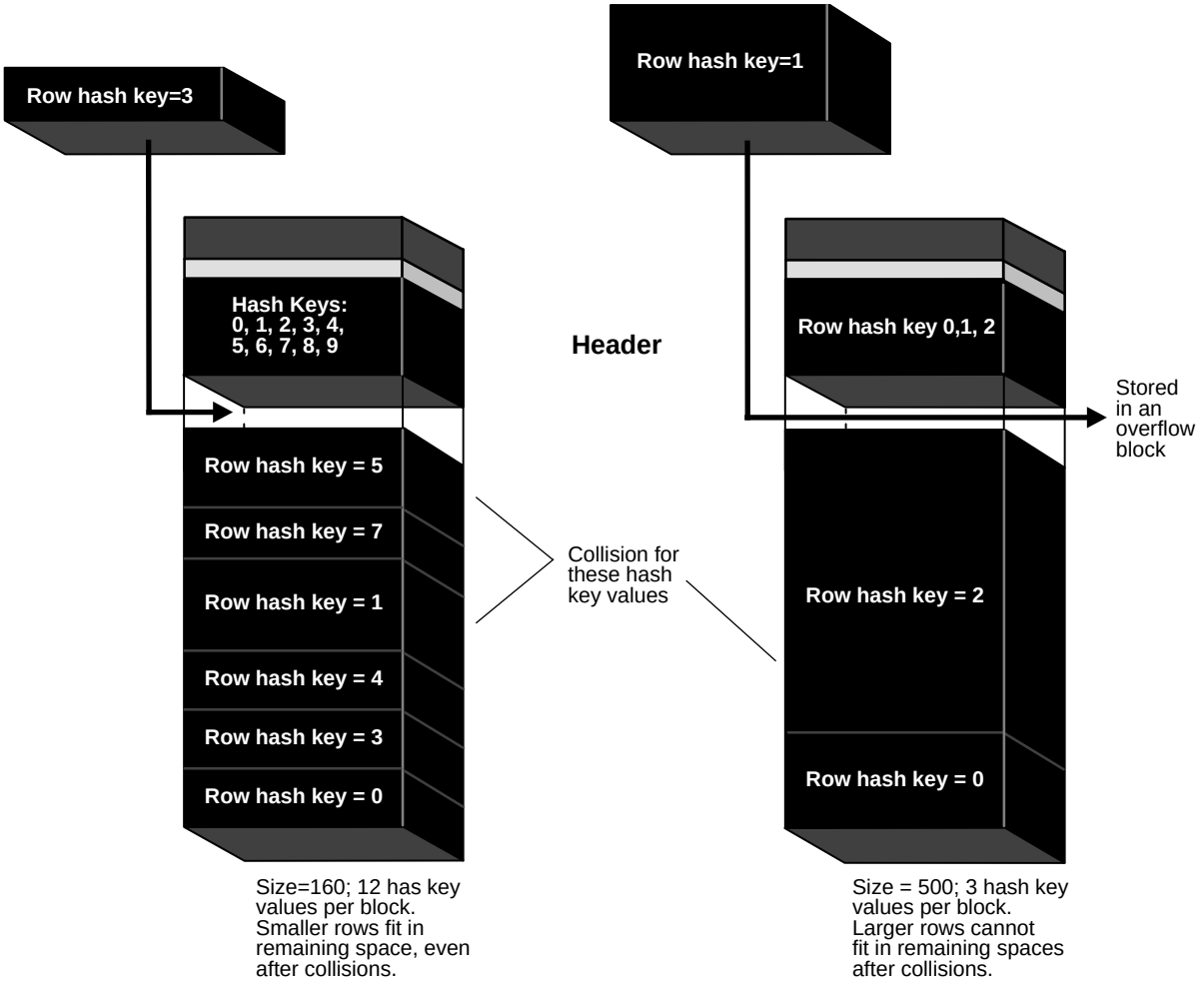
Allocation of Space for a Hash Cluster

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 8–11.

Figure 8–11 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 fit in the root block allocated for the hash key value and an overflow block is allocated.

9

Partitioned Tables and Indexes

Like to a double cherry, seeming parted,
But yet an union in partition;
Two lovely berries molded on one stem.

Wm. Shakespeare: *A Midsummer-Night's Dream*

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

- What Is Partitioning?
- Advantages of Partitioning
- Basic Partitioning Model
- Rules for Partitioning Tables and Indexes
- DML Partition Locks
- Maintenance Operations
- Managing Indexes
- Privileges for Partitioned Tables and Indexes
- Auditing for Partitioned Tables and Indexes
- SQL Extension: Partition-Extended Table Name

Attention: The features described in this chapter are available only if you have purchased Oracle8 Enterprise Edition with the Partitioning Option. See *Getting to Know Oracle8 and the Oracle8 Enterprise Edition* for information about the features and options available with Oracle8 Enterprise Edition.

Introduction to Partitioning

This section explains how partitioning can help you manage large tables and indexes in an Oracle database.

Note: Oracle only supports partitioning for tables and indexes; it does not support partitioning of clustered tables and their indexes, nor of snapshots.

What Is Partitioning?

Partitioning addresses the key problem of supporting very large tables and indexes by allowing you to decompose them into smaller and more manageable pieces called *partitions*.

Once partitions are defined, SQL statements can access and manipulate the partitions rather than entire tables or indexes. Partitions are especially useful in data warehouse applications, which commonly store and analyze large amounts of historical data.

All partitions of a table or index have the same logical attributes, although their physical attributes can be different. For example, all partitions in a table share the same column and constraint definitions; and all partitions in an index share the same index columns. However, storage specifications and other physical attributes such as PCTFREE, PCTUSED, INITRANS, and MAXTRANS can vary for different partitions of the same table or index.

Each partition is stored in a separate segment. Optionally, you can store each partition in a separate tablespace, which has the following advantages:

- You can contain the impact of damaged data.
- You can back up and recover each partition independently.
- You can balance I/O load by mapping partitions to disk drives.

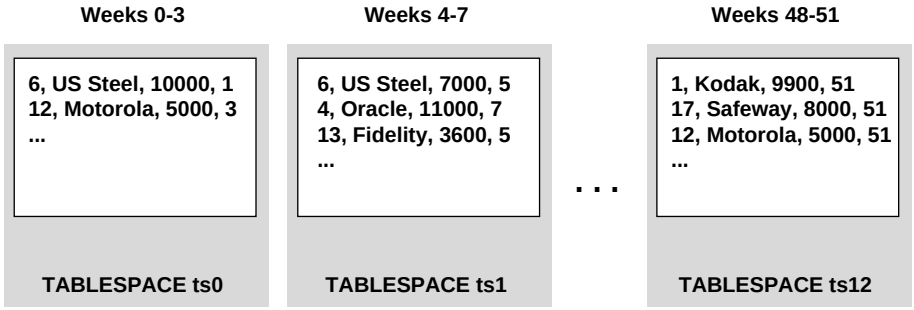
The section “Basic Partitioning Model” on page 9-11 provides more information about partitioning concepts.

Example of a Partitioned Table

In Figure 9–1, the **sales** table contains historical data divided by week number into 13 four-week partitions. This SQL statement creates the partitioned table:

```
CREATE TABLE sales ( acct_no         NUMBER(5),
                     acct_name       CHAR(30),
                     amount_of_sale  NUMBER(6),
                     week_no         INTEGER )
       PARTITION BY RANGE ( week_no ) ...
           (PARTITION sales1 VALUES LESS THAN ( 4 ) TABLESPACE ts0,
            PARTITION sales2 VALUES LESS THAN ( 8 ) TABLESPACE ts1,
            ...
            PARTITION sales13 VALUES LESS THAN ( 52 ) TABLESPACE ts12 );
```

Figure 9–1 SALES Table Partitioned by Week



Additional Infromation: For more examples of partitioned tables, see the *Oracle8 Administrator's Guide.*

Partition Pruning

The Oracle server incorporates the intelligence to explicitly recognize partitions. This knowledge is exploited in optimizing SQL statements to mark the partitions that need to be accessed, eliminating ("pruning") unnecessary partitions from access by those SQL statements.

For each SQL statement, 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 pruning can dramatically reduce the data volume, resulting in substantial improvements in query performance.

If the optimizer determines that the selection criteria used for pruning are satisfied by all the rows in the accessed partition, it removes those criteria from the predicate

list (WHERE clause) during evaluation in order to improve performance.

Partition pruning can eliminate index partitions even when the underlying table's partitions cannot be eliminated, if the index and table are partitioned on different columns. You can often improve the performance of operations on large tables by creating partitioned indexes which reduce the amount of data that your SQL statements need to access or modify.

The ability to prune unneeded partitions from SQL statements increases performance and availability for many purposes, including partition-level load, purge, backup, restore, reorganization, and index building.

Advantages of Partitioning

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

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

Very Large Databases (VLDBs)

A *Very Large Database (VLDB)* contains hundreds of gigabytes or even a few terabytes of data. Partitioning provides support for VLDBs that contain mostly structured data, rather than unstructured data. These VLDBs typically owe their size to the presence of a few very large data objects (tables and indexes) rather than to the presence of a very large number of data objects.

There are two major categories of VLDB:

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

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

Partitioning efficiently supports both OLTP VLDBs and DSS VLDBs.

Historical Databases

Historical databases are the most common type of DSS VLDB. A historical database contains two classes of tables, historical tables and enterprise tables.

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

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

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

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

Mission-Critical Databases

Mission-critical OLTP databases present special availability and performance problems even if they are not very large. For example, it may be necessary to perform

scheduled maintenance operations or recover a 10-gigabyte table in a very short period of time, perhaps an hour or less. Also, the DBA may need a degree of control over data placement that is hard to achieve when a table or index is spread over multiple drives.

Partitioning can increase the availability of mission-critical databases if critical tables and indexes are divided into partitions to reduce the maintenance windows, recovery times, and impact of failures. You can also improve access performance to a critical table or index by controllin performance parameters on a partition basis.

Reducing Downtime for Scheduled Maintenance

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

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

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

Partition Maintenance Operations

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

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

provided there are no interpartition stored constructs (global indexes and referential integrity constraints).

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

Partition Independence

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

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

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

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

Reducing Downtime Due to Data Failures

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

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

Storing partitions in separate tablespaces provides the following benefits:

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

DSS Performance

DSS queries on very large tables present special performance problems. An ad-hoc query that requires a table scan may take a long time, because it must inspect every row in the table; there is no way to identify and skip subsets of irrelevant rows. The

problem is particularly important for historical tables, for which many queries concentrate access on rows that were generated recently.

Partitions help solve this DSS performance problem. An ad-hoc query which only requires rows that correspond to a single partition (or range of partitions) can be executed using a partition scan rather than a table scan.

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

I/O Performance

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

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

Disk Striping: Performance versus Availability

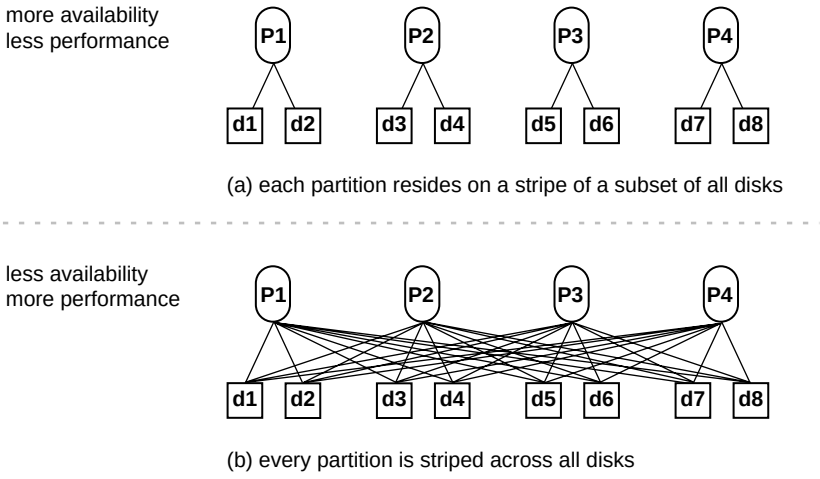
Disk striping and partitioning are both tools that can improve performance through the reduction of contention for disk arms. Which tool to use, or in which proportions to use them together, is an important issue to consider when physically designing databases. These issues should be considered not only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

Figure 9–2 shows the two extremes of combining partitioning and striping. Both (a) and (b) show four partitions spread across eight disks, but (a) stripes each partition onto its own pair of disks, whereas (b) stripes each partition onto all eight disks.

- The performance characteristics are better in (b), but if any single disk failure occurs, all partitions are adversely affected.
- The availability characteristics are better in (a), because failure of a single disk only affects one partition.

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

Figure 9–2 Partitions and Disk Striping



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

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

Partition Transparency

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

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

Manual Partitioning with Partition Views

Instead of using partitioned tables, you can build separate tables with identical templates and define a view that does a UNION of these tables. This is known as *manual partitioning*, and the view is known as a *partition view*.

Partition views were the only form of partitioning available in Oracle7 Release 7.3. They are not recommended for new applications in Oracle8. Partition views that were created for Oracle7 databases can be converted to partitioned tables by using the EXCHANGE PARTITION option of the ALTER TABLE command.

Note: Oracle8 supports partition views solely for backwards-compatibility with Oracle7 Release 7.3.

Additional Information: See *Oracle8 Migration* and the *Oracle8 Administrator's Guide* for instructions on converting partition views to partitioned tables.

The basic idea behind partition views is to divide the large table into multiple physical tables using a partitioning criterion (a WHERE clause or CHECK constraint), then glue the smaller tables together into a whole with a UNION ALL view. You can then define sets of "base indexes" with identical key specifications on the base tables, which provide indexing capabilities when the UNION ALL view is used. Partition views must be indexed to work properly.

Queries that use a key range to select from a partition view access only the base tables that lie within the key range. The optimizer can use separate execution plans for a partition view's base tables. (In contrast, the optimizer uses a single execution plan for all partitions in a partitioned table.)

Manual partitioning with partition views has a number of disadvantages:

- Configuration complexity

  The database administrator is responsible for correctly defining the base tables and indexes that correspond to partitions, and for maintaining these definitions. The equivalent of DDL operations that move data across partitions (split, move, and so on) must be implemented via Export/Import or SQL scripts.

- Lack of partition transparency

  Some SQL operations must be performed using the base tables rather than the UNION ALL view. For example, INSERT refers to a base table, and user code is needed to obtain the table name that appears in an INSERT statement.

- Lack of performance

  Some SQL operations on the UNION ALL view may perform badly because the optimizer does not take advantage of all the existing base indexes.

- Poor memory utilization

  A SQL compiled query operating on a UNION ALL view internally replicates descriptive information for all tables that support the view.

- DDL restrictions

  Global indexes and referential integrity constraints cannot be defined on the UNION ALL view.

- Load restrictions

  It is not possible to perform direct loads on a UNION ALL view.

Basic Partitioning Model

Partitioning is specified with options to the CREATE TABLE and CREATE INDEX statements. After creating a partitioned table or index, you can use ALTER TABLE or ALTER INDEX statements to modify its partitioning attributes. The partitioning syntax for CREATE TABLE and CREATE INDEX statements is very similar.

The CREATE TABLE statement specifies:

1. The logical attributes of the table, such as column and constraint definitions.
2. The physical attributes of the table.
   - If the table is non-partitioned, these are the real physical attributes of the segment associated with the table.
   - If the table is partitioned, these table-level attributes specify defaults for the individual partitions of the table.
3. For a partitioned table, there is also a *partition specification* which includes:
   - the table-level algorithm used to map rows to partitions
   - a list of partition descriptions, one for each partition in the table.

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

Datatype Restrictions For partitioned tables, the logical attributes have additional restrictions. Partitioned tables cannot have any columns with LONG or LONG RAW datatypes, LOB datatypes (BLOB, CLOB, NCLOB, or BFILE), or object types.

If a table (or index) is partitioned on a column that has the DATE datatype, its partition descriptions should use the TO_DATE format mask; otherwise partition pruning is not possible. See "The TO_DATE Format Mask" on page 9-16.

Bitmap Restrictions You can create bitmap indexes on partitioned tables, with the restriction that the bitmap indexes must be local to the partitioned table — they cannot be global indexes. (See "Index Partitioning" on page 9-22.)

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

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

- ALL_TAB_PARTITIONS, DBA_TAB_PARTITIONS, USER_TAB_PARTITIONS
- ALL_IND_PARTITIONS, DBA_IND_PARTITIONS, USER_IND_PARTITIONS
- ALL_PART_COL_STATISTICS, DBA_PART_COL_STATISTICS, USER_PART_COL_STATISTICS

Range Partitioning

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

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

and by the partitioning specifications for each individual partition:

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

where:

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

- *value_list* is an ordered list of values for the columns in *column_list*.
  - Each value in *value_list* must be either a literal or a TO_DATE() or RPAD() function with constant arguments. (See "The TO_DATE Format Mask" on page 9-16.)
  - The *value_list* contained in the partitioning specification for the ith partition defines an open (non-inclusive) upper bound for the partition, referred to as the *partition bound*.
  - The partition bound for the **i**th partition defines an open (non-inclusive) upper bound for the partition. The partition bound for the **i**th partition must compare less than the partition bound for the **(i+1)**th partition.

In the **i**th partition, all rows (or rows pointed to by index entries) have *partitioning keys* that compare less than the *partition bound* for that partition. Unless the **i**th partition is the first partition in the table or index, all of the partitioning keys in the **i**th partition also compare greater than or equal to the partition bound for the **(i-1)**th partition. (See "Partition Bounds and Partitioning Keys" on page 9-14 for more information about how partitioning keys are compared to partition bounds, and in particular how multicolumn partitioning keys are handled.)

For example, in the following table of four partitions (one for each quarter's sales), a row with **sale_year**=1997, **sale_month**=7, and **sale_day**=18 has partitioning key (1997, 7, 18), belongs in the third partition, and would be stored in tablespace **tsc**. A row with **sale_year**=1997, **sale_month**=7, and **sale_day**=1 has partitioning key (1997, 7, 1), and also belongs in the third partition, stored in tablespace **tsc**.

```
CREATE TABLE sales
    ( invoice_no NUMBER,
      sale_year  INT NOT NULL,
      sale_month INT NOT NULL,
      sale_day   INT NOT NULL )
  PARTITION BY RANGE (sale_year, sale_month, sale_day)
    ( PARTITION sales_q1 VALUES LESS THAN (1994, 04, 01)
        TABLESPACE tsa,
      PARTITION sales_q2 VALUES LESS THAN (1994, 07, 01)
        TABLESPACE tsb,
      PARTITION sales_q3 VALUES LESS THAN (1994, 10, 01)
        TABLESPACE tsc,
      PARTITION sales_q4 VALUES LESS THAN (1995, 01, 01)
        TABLESPACE tsd );
```

Partition Names

Every partition has a name, which must conform to the usual rules for naming schema objects and their parts. In particular:

- The name of a table partition must be unique among all the partitions belonging to the same parent table.
- The name of an index partition must be unique among all the partitions belonging to the same parent index.

You can rename a partition; however, you cannot create any synonyms on a partition name.

> **Additional Information:** See *Oracle8 SQL Reference* for information about the rules for naming schema objects.

Referencing a Partition

Partition names can optionally be referenced in DDL and DML statements and in utility statements like Import/Export and SQL*Loader. They always appear in context with the name of their parent table or index and they are never qualified by a schema name. (The schema name can be used to qualify the parent table or index.)

For example:

```
ALTER TABLE admin.patient_visits DROP PARTITION pv_dec92
```

See "SQL Extension: Partition-Extended Table Name" on page 9-42 for more information about referencing partitions in SQL statements.

Partition Bounds and Partitioning Keys

This section describes how a row's partitioning key is compared with a set of upper and lower bounds to determine which partition the row belongs in.

Partition Bounds

Every table and index partition has a non-inclusive *upper bound*, which is specified by the VALUES LESS THAN clause. Every partition except the first partition also has a *lower bound* (inclusive), which is specified by the VALUES LESS THAN on the next-lower partition.

The partition bounds collectively define an ordering of the partitions in a table or index. The "first" partition is the partition with the lowest VALUES LESS THAN clause, and the "last" or "highest" partition is the partition with the highest VALUES LESS THAN clause.

If you attempt to insert a row into a table and the row's partitioning key is greater than or equal to the partition bound for the highest partition in the table, the insert will fail.

Partitioning Keys

A *partitioning key* consists of an ordered list of up to 16 columns. A row's partitioning key is an ordered list of its values for the partitioning columns.

A partitioning key may not contain the LEVEL, ROWID, or MLSLABEL pseudocolumn or a column of type ROWID.

When comparing character values in partitioning keys and partition bounds, characters are compared according to their binary values. However, if a character consists of more than one byte, Oracle compares the binary value of each byte, not of the character.

The comparison also uses the comparison rules associated with the column data type (for example, blank-padded comparison is done for the ANSI CHAR data type). The NLS parameters, specifically the initialization parameters NLS_SORT and NLS_LANGUAGE and the environment variable NLS_LANG, have no effect on the comparison.

MAXVALUE

You can specify the keyword MAXVALUE for any value in the partition bound *value_list*. This keyword represents a virtual "infinite" value that sorts higher than any other value for the data type, including the null value.

For example, you might partition the **office** table on **state** (a CHAR(10) column) into three partitions with the following partition bounds:

- VALUES LESS THAN ('I'): Contains states whose names start with A through H.
- VALUES LESS THAN ('S'): Contains states whose names start with I through R.
- VALUES LESS THAN (MAXVALUE): Contains states whose names start with S through Z, plus special codes for non-U.S. regions.

Nulls

NULL cannot be specified as a value in a partition bound *value_list*. An empty string also cannot be specified as a value in a partition bound *value_list*, because it is treated as NULL within the database server.

For the purpose of assigning rows to partitions, Oracle sorts nulls greater than all other values except MAXVALUE. Nulls sort less than MAXVALUE.

This means that if a table is partitioned on a nullable column, and the column is to contain nulls, then the highest partition should have a partition bound of MAXVALUE for that column. Otherwise the rows that contain nulls will map above the highest partition in the table and the insert will fail.

The TO_DATE Format Mask

If the partition key includes a column that has the DATE datatype, you must specify partition bounds using the TO_DATE() format mask; otherwise partition elimination ("pruning") will not work.

For example, you might create the **sales** table using a DATE column:

```
CREATE TABLE sales
    ( invoice_no NUMBER,
      sale_date DATE NOT NULL )
  PARTITION BY RANGE (sale_date)
    ( PARTITION sales_q1
        VALUES LESS THAN (TO_DATE('94-04-01','YY-MM-DD'))
        TABLESPACE tsa,
      PARTITION sales_q2
        VALUES LESS THAN (TO_DATE('94-07-01','YY-MM-DD'))
        TABLESPACE tsb,
      PARTITION sales_q3
        VALUES LESS THAN (TO_DATE('94-10-01','YY-MM-DD'))
        TABLESPACE tsc,
      PARTITION sales_q4
        VALUES LESS THAN (TO_DATE('95-01-01','YY-MM-DD'))
        TABLESPACE tsd );
```

You also need to use the TO_DATE() format mask when you query or modify data in the **sales** table, for example:

```
SELECT * FROM sales
  WHERE sale_date < TO_DATE('94-06-15','YY-MM-DD');
```

Multicolumn Partitioning Keys

When a table or index is partitioned on multiple columns, each partition bound and partitioning key is a list (or vector) of values. In this case, the keys are ordered according to ANSI SQL2 vector comparison rules (this is also the way multicolumn index keys are ordered in Oracle).

For vectors V1 and V2 which contain the same number of values, Vx[i] is the ith value in Vx. Assuming that V1[i] and V2[i] have compatible data types:

- V1 = V2 if and only if V1[i] = V2[i] for all i.
- V1 < V2 if and only if V1[i] = V2[i] for all i < n and V1[n] < V2[n] for some n.
- V1 > V2 if and only if V1[i] = V2[i] for all i < n and V1[n] > V2[n] for some n.

That is, if you want to know if a partitioning key PK is less than or equal to partition bound PB, you compare corresponding values in PK and PB until you find a pair that is not equal and that pair decides.

For example, if the partition bound for partition P is (7, 5, 10) and the partition bound for the next lowest partition is (6, 7, 3) then:

- Keys (6, 9, 11) and (7, 3, 15) belong in partition P, because:
  - key (6, x, x) is less than (7, x, x), and (6, 9, x) is greater than (6, 7, x)
  - key (7, 3, x) is less than (7, 5, x) and greater than (6, 7, x)
- Keys (6, 5, 0) and (7, 5, 11) belong in other partitions.

If MAXVALUE appears as an element of a partition bound *value_list*, then the values of all the following elements are irrelevant. For example, a partition bound of (10, MAXVALUE, 5) is equivalent to a partition bound of (10, MAXVALUE, 6) or to a partition bound of (10, MAXVALUE, MAXVALUE).

Multicolumn partitioning keys are useful when the primary key for the table contains multiple columns, but rows are not distributed evenly over the most significant column in the key. For example, suppose that the **supplier_parts** table contains information about which suppliers provide which parts, and the primary key for the table is (**suppnum**, **partnum**). It is not sufficient to partition on **suppnum** because some suppliers provide hundreds of thousands of parts, while others provide only a few specialty parts. Instead, you can partition the table on (**suppnum**, **partnum**).

Multicolumn partitioning keys are also useful when you represent a date as three CHAR columns instead of a DATE column.

Implicit Constraints Imposed by Partition Bounds

If you specify a partition bound other than MAXVALUE for the highest partition in a table, this imposes an implicit CHECK constraint on the table. This constraint is not recorded in the data dictionary (but the partition bound itself is recorded).

Equipartitioning

Two tables or indexes are *equipartitioned* if they have identical logical partitioning attributes. They do not have to be the same type of schema object; for example, a table and an index can be equipartitioned.

If A and B are partitioned tables or indexes, where A[i] is the ith partition in A and B[i] is the ith partition in B, then A and B are equipartitioned if all of the following are true:

- They have the same number of partitions N.
- They have the same number of partitioning columns M.
- For every 1 <= i <= N, A[i] and B[i] have the same partition bound.

If Apcol[i] is the ith partitioning column in A and Bpcol[i] is the ith partitioning column in B, then the following must also be true:

- For 1 <= i <= M, Apcol[i] and Bpcol[i] have the same data type, including length, precision, and scale.

A[i] and B[i] may differ in their physical attributes; in particular they do not have to reside in the same tablespace.

Equipartitioning is important to consider when designing the database.

- It reduces the downtime and the amount of data that is unavailable during partition maintenance operations and tablespace recovery operations. For example, if a table and its indexes are equipartitioned then the effect of splitting a partition is limited to one table partition and the corresponding index partitions, but if a table has an index that is not equipartitioned then splitting one partition of the table makes it necessary to reorganize the entire index.
- Equipartitioning between tables improves execution plans by reducing the number of large sorts and joins that have to be performed. (This improvement applies only to serial execution of SQL statements.)
- It makes tablespace incomplete recovery (point-in-time recovery) on related subsets of data easier. For example, you might equipartition a table and its primary key index, or a parent table and a child table. You could then recover corresponding partitions to a point in time.

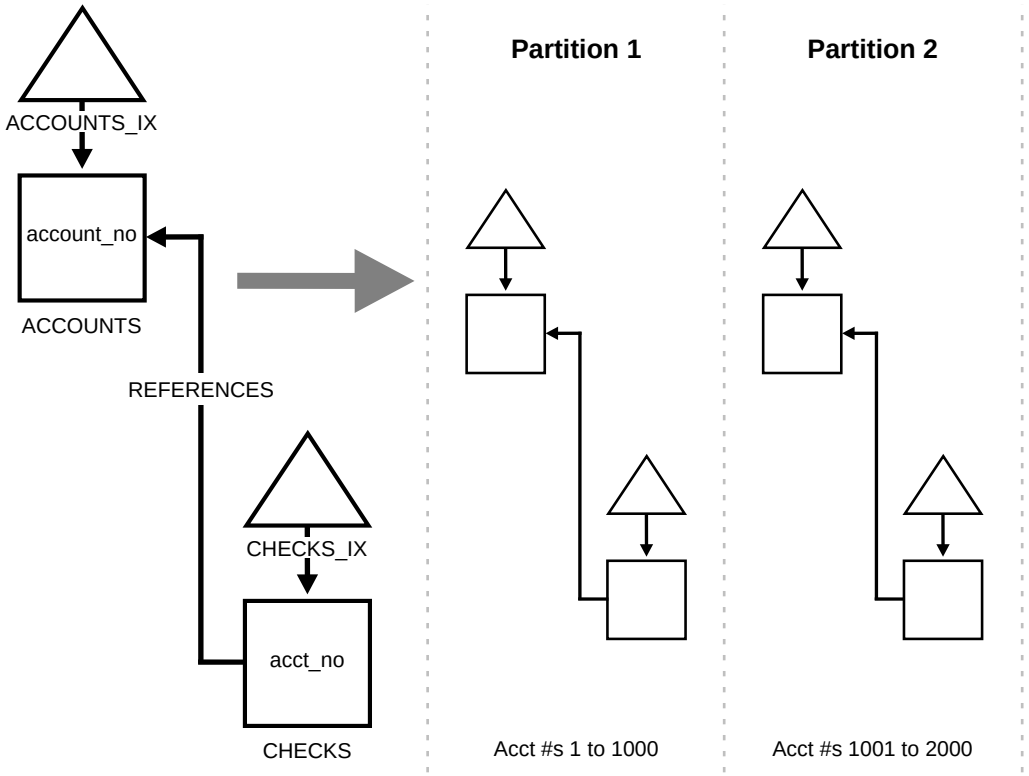
Example of Equipartitioning

Figure 9–3 shows four logically related schema objects that are equipartitioned:

- ACCOUNTS is a table with two partitions which is range-partitioned on column ACCOUNT_NO. The first partition contains account numbers up to 1000. The second partition contains account numbers up to 2000.
- ACCOUNTS_IX is an index on column ACCOUNT_NO in the ACCOUNTS table. Like the table, the index is range-partitioned on ACCOUNT_NO into two partitions, which have the same partition bounds as partitions of ACCOUNTS.
- CHECKS is a table with two partitions which is range-partitioned on column ACCT_NO. Its partitions have the same partition bounds as partitions of the ACCOUNTS table. ACCT_NO is a foreign key that references ACCOUNT_NO in ACCOUNTS.
- CHECKS_IX is an index on columns (ACCT_NO, CHECK_NO) in CHECKS. It is range-partitioned on ACCT_NO into two partitions, which have the same partition bounds as partitions of ACCOUNTS.

The logical relationship between the four schema objects is shown on the left in Figure 9–3; the physical partitioning is shown on the right. (Triangles represent indexes and rectanges represent tables.)

Figure 9–3 Equipartitioned Tables and Indexes



Rules for Partitioning Tables and Indexes

This section describes the rules for creating partitioned tables and indexes and the physical attributes of partitions.

Table Partitioning

The rules for partitioning tables are simple:

- A table can be range-partitioned, provided that:
  - It is not part of a cluster.
  - It does not contain LOBs, LONG or LONG RAW datatypes, or object types.
  - It is not an index-organized table.
- You can mix partitioned and non-partitioned indexes with partitioned and non-partitioned tables:
  - A partitioned table can have partitioned and/or non-partitioned indexes.
  - A non-partitioned table can have partitioned and/or non-partitioned indexes. (Only global indexes can be created on non-partitioned tables — see “Global Indexes” on page 9-25.)

Physical Attributes of Table Partitions

Default physical attributes are initially specified when the CREATE TABLE statement creates a partitioned table. Since there is no segment corresponding to the partitioned table itself, these attributes will only be used in derivation of physical attributes of member partitions. Default physical attributes can later be modified using ALTER TABLE MODIFY DEFAULT ATTRIBUTES.

Physical attributes of table partitions created by CREATE TABLE and ALTER TABLE ADD PARTITION are determined as follows:

- Values of physical attributes specified (explicitly or by default) for the base table are used whenever the value of a corresponding partition attribute was not specified.

Physical attributes of an existing table partition may be modified by ALTER TABLE MOVE PARTITION and ALTER TABLE MODIFY PARTITION. Resulting attributes are determined as follows:

- Values of physical attributes of the partition before the statement was issued are used whenever a new value was not specified.

Note: ALTER TABLE MOVE PARTITION may be used to change the tablespace in which a partition resides.

Physical attributes of table partitions created by ALTER TABLE SPLIT PARTITION are determined as follows:

- Values of physical attributes of the partition being split are used whenever a new value was not specified. (This also applies to global index split — missing attributes are inherited from the index partition being split.)

Physical attributes of all partitions of a table may be modified by ALTER TABLE, for example, ALTER TABLE *tablename* NOLOGGING changes the logging mode of all partitions of *tablename* to NOLOGGING.

Index Partitioning

The rules for partitioning indexes are similar to those for tables:

- An index can be range-partitioned with these exceptions:
  - The index is not a cluster index.
  - The index is not defined on a clustered table.
  - A bitmap index on a partitioned table must be a local index.
- You can mix partitioned and non-partitioned indexes with partitioned and non-partitioned tables:
  - A partitioned table can have partitioned and/or non-partitioned indexes.
  - A non-partitioned table can have partitioned and/or non-partitioned B*-tree indexes.
  - Bitmap indexes on non-partitioned tables cannot be range-partitioned.

However, partitioned indexes are more complicated than partitioned tables because there are four types of range-partitioned indexes: local prefixed, local non-prefixed, global prefixed, and global non-prefixed. These types are described below. Oracle supports three of the four types (global non-prefixed indexes are not useful in real applications).

Local Indexes

In a *local index*, all keys in a particular index partition refer only to rows stored in a single underlying table partition. A local index is created by specifying the LOCAL attribute.

Oracle constructs the local index so that it is equipartitioned with the underlying table. Oracle range-partitions the index on the same columns as the underlying table, creates the same number of partitions, and gives them the same partition bounds as corresponding partitions of the underlying table. Oracle also maintains the index partitioning automatically as partitions in the underlying table are added, dropped, or split. This ensures that the index remains equipartitioned with the table.

Equipartitioning a table and its index has the following advantages:

- Only one index partition is affected when a maintenance operation (other than SPLIT PARTITION) is performed on an underlying table partition.
  - The duration of a partition maintenance operation remains proportional to partition size if the partitioned table has only local indexes.
  - Local indexes support partition independence.
  - Local indexes support smooth roll-out of old data and roll-in of new data in historical tables.
- Oracle can take advantage of the fact that a local index is equipartitioned with the underlying table to generate better query access plans.
- Local indexes simplify the task of tablespace incomplete recovery. In order to recover a partition of a table to a point in time, you must also recover the corresponding index entries to the same point in time. The only way to accomplish this is with a local index; then you can recover the corresponding table and index partitions together.

Local Prefixed Indexes

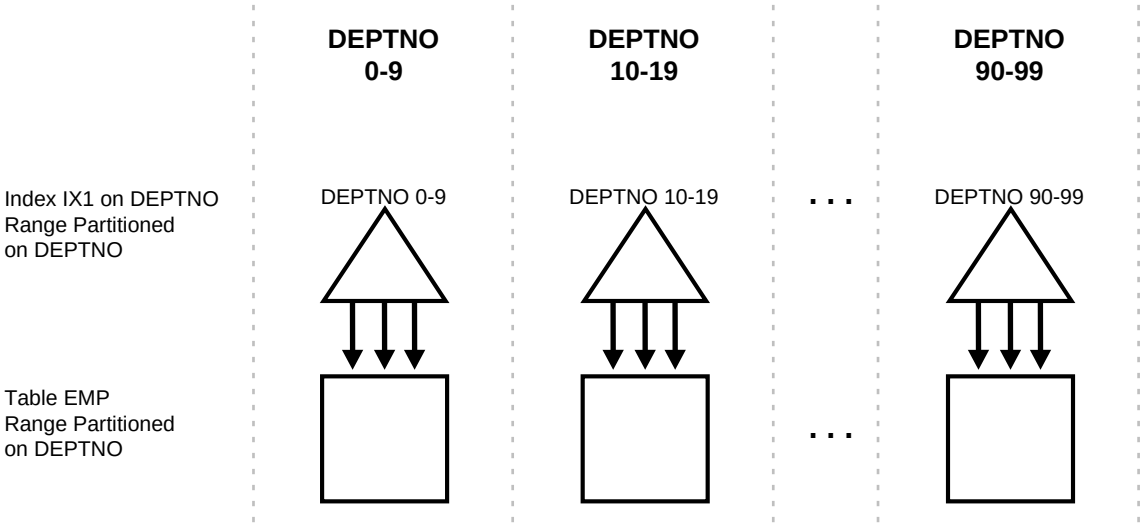
A local index is *prefixed* if it is partitioned on a left prefix of the index columns.

For example, if the **sales** table and its local index **sales_ix** are partitioned on the **week_num** column, then index **sales_ix** is *local prefixed* if it is defined on the columns (**week_num,xaction_num**). On the other hand, if index **sales_ix** is defined on column **product_num** then it is not prefixed.

Figure 9–4 shows another example of a local prefixed index.

Local prefixed indexes can be unique or non-unique.

Figure 9–4 Local Prefixed Index



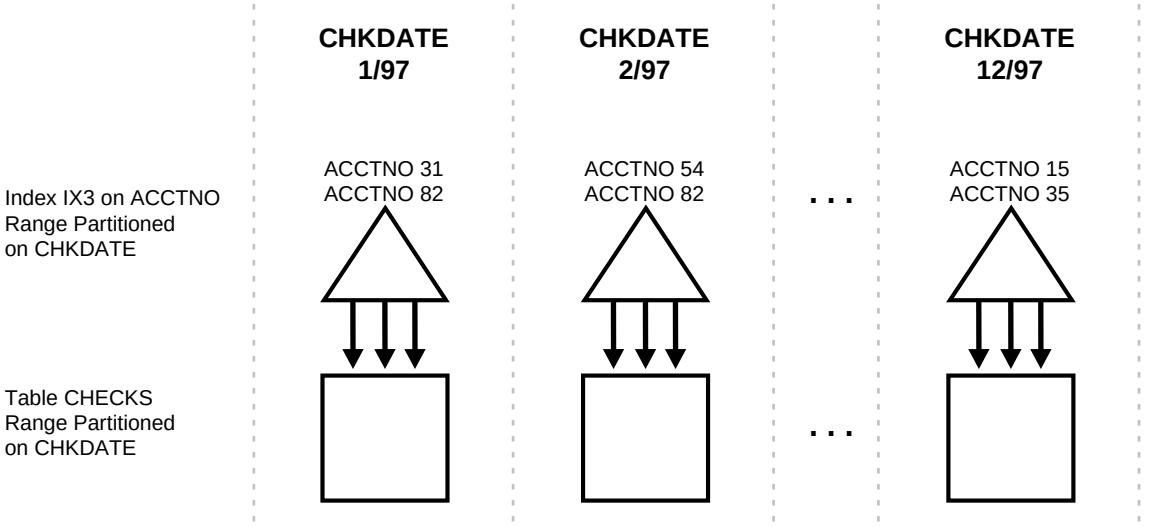
Local Non-Prefixed Indexes

A local index is *non-prefixed* if it is not partitioned on a left prefix of the index columns.

You cannot have a unique local non-prefixed index unless the index key is a subset of the partitioning key.

Figure 9–5 shows an example of a local non-prefixed index.

Figure 9–5 Local Non-Prefixed Index



Global Indexes

In a *global index*, the keys in a particular index partition may refer to rows stored in more than one underlying table partition. A global index is created by specifying the GLOBAL attribute (this is the default). The user is responsible for defining the initial partitioning of a global index at creation and for maintaining the partitioning over time. Index partitions can be dropped or split as necessary.

Normally, a global index is not equipartitioned with the underlying table. There is nothing to prevent an index from being equipartitioned with the underlying table, but Oracle does not take advantage of the equipartitioning when generating query plans or executing partition maintenance operations. So an index that is equipartitioned with the underlying table should be created as LOCAL.

A global index contains (conceptually) a single B*-tree with entries for all rows in all partitions. Each index partition may contain keys that refer to many different partitions in the table.

The highest partition of a global index must have a partition bound all of whose values are MAXVALUE. This insures that all rows in the underlying table can be represented in the index.

A global index is *prefixed* if it is partitioned on a left prefix of the index columns. (See Figure 9–6 for an example.) A global index is *non-prefixed* if it is not partitioned

on a left prefix of the index columns. Oracle does not support global non-prefixed indexes.

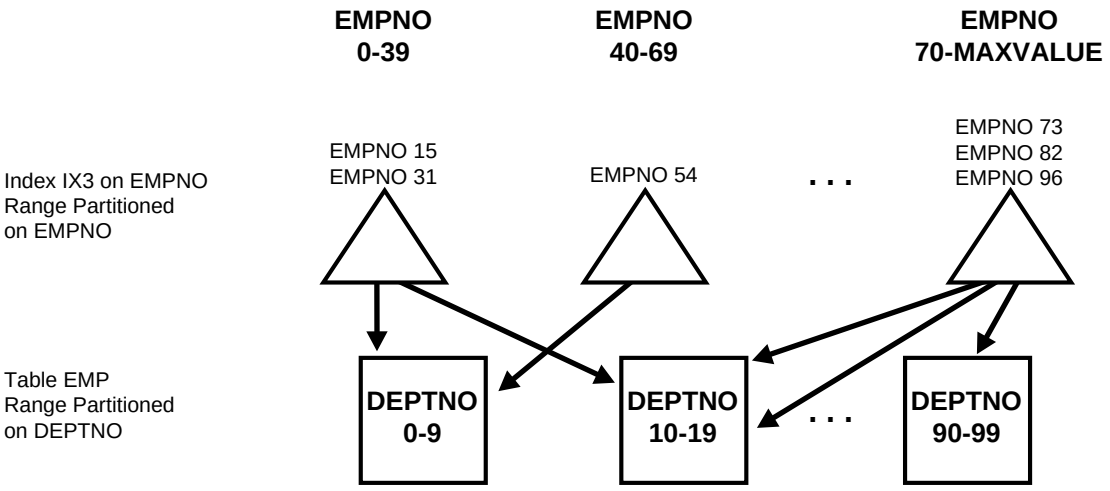
Global prefixed indexes can be unique or non-unique.

Global indexes are harder to manage than local indexes:

- When the data in an underlying table partition is moved or removed (SPLIT, MOVE, DROP, or TRUNCATE), all partitions of a global index are affected. Consequently global indexes cause partition maintenance (including rebuilds of global indexes or index partitions) to have duration proportional to table size rather than partition size, and they do not support partition independence.
- When an underlying table partition is recovered to a point in time, all corresponding entries in a global index must be recovered to the same point in time. Because these entries may be scattered across all partitions of the index (mixed in with entries for other partitions that are not being recovered), there is no way to accomplish this except by recreating the entire global index.

Non-partitioned indexes are treated as global prefixed indexes.

Figure 9–6 Global Prefixed Index



Summary of Partitioned Index Types

Table 9–1 summarizes the three types of partitioned indexes that Oracle supports.

- If an index is *local*, it is equipartitioned with the underlying table; otherwise it is *global*.
- A *prefixed* index is partitioned on a left prefix of the index columns; otherwise it is *non-prefixed*.

Table 9–1 Types of Partitioned Indexes

Type of Index	Index Equi-partitioned with Table	Index Partitioned on Left Prefix of Index Columns	UNIQUE Attribute Allowed	Example		
				Table Partitioned On Column	Index Columns	Index Partitioned On Column
Local Prefixed	Yes	Yes	Yes	A	A,B	A
Local Non-Prefixed	Yes	No	Yes[1]	A	B	A
Global Prefixed	No[2]	Yes	Yes	A	B	B
Global Non-Prefixed[3]	—	—	—	—	—	—

[1] For a unique local non-prefixed index, the index key must be a subset of the partitioning key.

[2] Although a global partitioned index may be equipartitioned with the underlying table, Oracle does not take advantage of the partitioning or maintain equipartitioning after partition maintenance operations such as DROP or SPLIT PARTITION.

[3] This type of index is not supported.

Importance of Non-Prefixed Indexes

Non-prefixed indexes are particularly useful in historical databases. In a table containing historical data it is common for an index to be defined on one column to support the requirements of fast access by that column, but partitioned on another column (the same column as the underlying table) to support the time interval for rolling out old data and rolling in new data.

Consider the **sales** table presented in Figure 9–1 (“SALES Table Partitioned by Week” on page 9-3). It contains a year’s worth of data, divided into 13 partitions. It is range partitioned on **week_no**, four weeks to a partition. You might create a non-prefixed local index **sales_ix** on **sales**. The **sales_ix** index is defined on **acct_no** because there are queries that need fast access to the data by account number. However it is partitioned on **week_no** to match the **sales** table. Every four weeks the oldest partitions of **sales** and **sales_ix** are dropped and new ones are added.

Performance Implications of Prefixed and Non-Prefixed Indexes

It is more expensive to scan a non-prefixed index than to scan a prefixed index.

If an index is prefixed (either local or global) and Oracle is presented with a predicate involving the index columns, then partition pruning can restrict application of the predicate to a subset of the index partitions.

For example, in Figure 9–4 (“Local Prefixed Index” on page 9-24) if the predicate is DEPTNO=15, the optimizer knows to apply the predicate only to the second partition of the index. (If the predicate involves a bind variable, the optimizer will not know exactly which partition but it may still know there is only one partition involved, in which case at run time only one index partition will be accessed.)

When an index is non-prefixed Oracle often has to apply a predicate involving the index columns to all N index partitions. This is required to look up a single key, or to do an index range scan. For a range scan, Oracle must also combine information from N index partitions. For example, in Figure 9–5 (“Local Non-Prefixed Index” on page 9-25) a local index is partitioned on CHKDATE with an index key on ACCTNO. If the predicate is ACCTNO=31, Oracle probes all 12 index partitions.

Of course, if there is also a predicate on the partitioning columns then multiple index probes might not be necessary. Oracle takes advantage of the fact that a local index is equipartitioned with the underlying table to prune partitions based on the partition key. For example, if the predicate in Figure 9–5 is CHKDATE<3/97, Oracle only has to probe two partitions.

So for a non-prefixed index, if the partition key is a part of the WHERE clause (but not of the index key) the optimizer determines which index partitions to probe based on the underlying table partition.

When many queries and DML statements using keys of local, non-prefixed, indexes have to probe all index partitions, this effectively reduces the degree of partition independence provided by such indexes.

Guidelines for Partitioning Indexes

When deciding how to partition indexes on a table, you must consider the mix of applications that need to access the table. There is a trade-off between performance on the one hand and availability and manageability on the other.

Here are some of the guidelines you should consider:

- For OLTP applications:
  - Global indexes and local prefixed indexes provide better performance than local non-prefixed indexes because they minimize the number of index

partition probes.

  - Local indexes support more availability when there are partition maintenance operations on the table. Local non-prefixed indexes are very useful for historical databases.

- For DSS applications, local non-prefixed indexes can improve performance because many index partitions can be scanned in parallel by range queries on the index key.

  For example, a query using the predicate "ACCTNO between 40 and 45" on the table CHECKS of Figure 9–5 ("Local Non-Prefixed Index" on page 9-25) causes parallel scans of all the partitions of the non-prefixed index IX3. On the other hand, a query using the predicate "DEPTNO between 40 and 45" on the table DEPTNO of Figure 9–4 ("Local Prefixed Index" on page 9-24) cannot be parallelized because it accesses a single partition of the prefixed index IX1.

- For historical tables, indexes should be local if possible. This limits the impact of regularly scheduled drop partition operations.

- Unique indexes on columns other than the partitioning columns must be global because unique local non-prefixed indexes whose key does not contain the partitioning key are not supported.

Physical Attributes of Index Partitions

Default physical attributes are initially specified when a CREATE INDEX statement creates a partitioned index. Since there is no segment corresponding to the partitioned index itself, these attributes are only used in derivation of physical attributes of member partitions. Default physical attributes can later be modified using ALTER INDEX.

Physical attributes of partitions created by CREATE INDEX are determined as follows:

- Values of physical attributes specified (explicitly or by default) for the index are used whenever the value of a corresponding partition attribute was not specified. Handling of the TABLESPACE attribute of partitions of a LOCAL index constitutes an important exception to this rule in that in the absence of a user-specified TABLESPACE value, that of the corresponding partition of the underlying table will be used.

Physical attributes (other than TABLESPACE, as explained above) of partitions of local indexes created in the course of processing ALTER TABLE ADD PARTITION are set to the default physical attributes of each index.

Physical attributes (other than TABLESPACE, as explained above) of index partitions created by ALTER TABLE SPLIT PARTITION are determined as follows:

- Values of physical attributes (other than TABLESPACE, as explained above) of the index partition being split are used.

Physical attributes of an existing index partition can be modified by ALTER INDEX MODIFY PARTITION and ALTER INDEX REBUILD PARTITION. Resulting attributes are determined as follows:

- Values of physical attributes of the partition before the statement was issued are used whenever a new value was not specified. Note that ALTER INDEX REBUILD PARTITION can be used to change the tablespace in which a partition resides.

Physical attributes of global index partitions created by ALTER INDEX SPLIT PARTITION are determined as follows:

- Values of physical attributes of the partition being split are used whenever a new value is not specified.

Physical attributes of all partitions of an index may be modified by ALTER INDEX, for example, ALTER INDEX *indexname* NOLOGGING changes the logging mode of all partitions of *indexname* to NOLOGGING.

DML Partition Locks

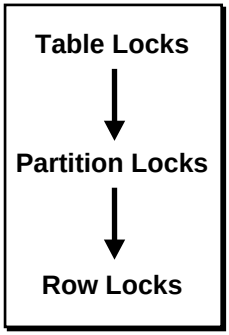
DML table locks synchronize DML statements (INSERT, UPDATE, and DELETE) with DDL statements and LOCK TABLE statements. DML table locks also synchronize DDL and LOCK TABLE statements among themselves.

In order to provide *partition independence* for DDL and utility operations, Oracle supports *DML partition locks*. Partition independence allows you to perform DDL and utility operations on selected partitions without quiescing activity on other partitions.

The purpose of a partition lock is to protect the data in an individual partition while multiple users are accessing that partition or other partitions in the table concurrently.

Partition locks fall between table locks and row locks in the DML locking hierarchy, as shown in Figure 9–7.

Figure 9–7 DML Locking Hierarchy



Partition locks can be acquired in the same modes as table locks: Share (S), Exclusive (X), Row Share (SS), Row Exclusive (SX), and Share Row Exclusive (SSX).

Performance Considerations for Oracle Parallel Server

Introducing an extra level of DML locking may affect the performance of short transactions in the Oracle Parallel Server environment because extra messages are sent to the Distributed Lock Manager.

To improve performance in the Oracle Parallel Server environment, you can turn off DML locking on selected tables with the ALTER TABLE DISABLE TABLE LOCK statement, which disables both table and partition DML locks. DDL statements are not allowed when DML locking is disabled.

> **Additional Information:** See *Oracle8 Parallel Server Concepts and Administration*.

Maintenance Operations

This section covers the following topics:

- Partition Maintenance Operations
- Managing Indexes
- Privileges for Partitioned Tables and Indexes
- Auditing for Partitioned Tables and Indexes

For the purposes of this chapter, a *maintenance operation* is a DDL statement or a utility (like Export, Import, SQL*Loader) that alters the definition of a table or index and/or does bulk load or unload of data.

Most maintenance operations on non-partitioned tables and indexes also work on partitioned tables and indexes. For example, DROP TABLE can drop a partitioned table, and Export can export a partitioned table. However, some maintenance operations must be performed on individual partitions rather than the whole partitioned table or index. For example, ALTER TABLE ALLOCATE EXTENT cannot be used for a partitioned table; instead, you use ALTER TABLE MODIFY PARTITION ALLOCATE EXTENT for the partition or partitions that need new extents.

Maintenance operations are considered fast if their expected duration is not affected by the size (number of records) of the schema objects they operate upon. Fast maintenance operations result only in dictionary and segment header changes, and do not cause data scans and data updates. They are expected to complete in a short time (order of seconds). For example, RENAME is a fast operation while CREATE INDEX is not a fast operation.

Partition Maintenance Operations

A *partition maintenance operation* modifies one partition of a partitioned table or index. For example, you might add a new partition to an existing table, or you might move a partition to a different tablespace for better I/O load balancing, or you might load a partition.

Some partition maintenance operations are planned events. For example, in a historical database, the database administrator (DBA) periodically drops the oldest partitions from the database and adds a set of new partitions. This drop and add operation occurs on a regularly scheduled basis. Another example of a planned maintenance operation is a periodic Export/Import to recluster data and reduce fragmentation.

Other partition maintenance operations are unplanned events, required to recover from application or system problems. For example, unexpected transaction activity may force the DBA to split a partition to rebalance I/O load, or the DBA may need to rebuild one or more index partitions.

The partition maintenance operations are:

- Add a table partition to an existing table
- Modify a partition — change the physical attributes of a partition
- Move a table partition — move it to another tablespace, or recluster it, or change any of its parameters (including any of its create-time parameters)

- Rename a partition
- Drop a partition
- Truncate a table partition (with or without reclaiming space)
- Split an existing partition into two partitions
- Load data into one table partition
- Export data from one table partition
- Import a table partition
- Rebuild an index partition

Concurrency Model for Maintenance Operations

The concurrency model described in this section defines when it is possible to run more than one DDL and utility operation on the same schema object at the same time. It also defines which query and DML operations can be run concurrently with DDL and utility operations.

The model applies to all DDL statements. It also applies to utilities like SQL*Loader.

One-Step and Three-Step Operations There are two types of maintenance operations, one-step and three-step.

One-step operations:

- These operations DML lock the affected table in Exclusive (X) mode. Index operations lock the underlying table. They also hold Exclusive dictionary locks for the duration of the operation.
- These operations are either fast (for example, ALTER TABLE ADD PARTITION) or they offer no possibility of running other operations concurrently (for example, ALTER TABLE ADD column).
- All index operations are one-step except CREATE INDEX and ALTER INDEX REBUILD (and ALTER INDEX DROP / SPLIT PARTITION, if the global partition being dropped or split is Usable).
- All Oracle DDL statements are one-step except CREATE INDEX and MOVE, SPLIT, REBUILD. or EXCHANGE PARTITION.

Three-step operations:

- These operations acquire less restrictive DML locks on the affected table. They lock only one partition in Exclusive (X) mode, or if they lock the entire table,

they lock it in an S or SS or SX mode.

- These operations consist of three steps:
  - Step 1: read dictionary while holding Share dictionary locks. Step 1 takes a short time (seconds). At the end of this step, the appropriate DML locks are acquired, then the dictionary locks are released.
  - Step 2: scan or update table or index records. Step 2 may take a long time (minutes or hours).
  - Step 3: update dictionary while holding Exclusive dictionary locks. Step 3 takes a short time (seconds).
- These operations are long running, but they allow other operations to run concurrently. Exactly which operations can run concurrently depends on the specific DML locks acquired by the statement, as explained below.
- The following operations are three-step:
  - ALTER TABLE MOVE PARTITION, ALTER TABLE SPLIT PARTITION, ALTER TABLE EXCHANGE PARTITION, Direct Path Load Table Partition: These statements lock the base table in Row Exclusive (SX) mode and the partition in Exclusive (X) mode.
  - CREATE INDEX and ALTER INDEX REBUILD PARTITION (for a global index): These statements lock the underlying table in Shared (S) mode.
  - ALTER INDEX REBUILD PARTITION (for a local index): This statement locks the underlying table in Row Share (SS) mode and the underlying table partition in Shared (S) mode.
  - ALTER TABLE MODIFY PARTITION REBUILD UNUSABLE LOCAL INDEXES: This statement locks the underlying table in Row Share (SS) mode and the underlying table partition in Shared (S) mode.

Finally, some operations may follow either one-step or three-step protocol:

- This group consists of ALTER TABLE DROP PARTITION and ALTER TABLE TRUNCATE PARTITION.

  If the table being altered has no global indexes defined on it, or if it is referenced by enabled referential constraints, statements in this group execute using the one-step protocol and they are fast. Otherwise, they execute using the three-step protocol. In the latter case the base table is locked in Row Exclusive (SX) mode and the partition is locked in Exclusive (X) mode.

- ALTER INDEX SPLIT PARTITION (allowed for global indexes only).

  If the partition to be split is USABLE, the statement follows the 3-step protocol, and partitions resulting from SPLIT are USABLE. If, on the other hand, the partition being split is UNUSABLE, the operation follows the 1-step protocol, and resulting partitions are also marked UNUSABLE.

- ALTER INDEX DROP PARTITION (allowed for global indexes only).

  If the partition to be dropped is USABLE, the statement follows the three-step protocol; otherwise it follows the one-step protocol.

Conventional Path SQL*Loader and Import use SQL INSERT so they are classified as DML operations for the purposes of the model. Export uses SQL SELECT so it is classified as a query operation.

Operations That Can Run Concurrently The rules in this section can be derived from the definitions of one-step and three-step operations.

While a one-step operation is in progress:

- You can run queries on the table.
- You cannot run any other operation (DDL, utility, or DML).

Since queries (READ operations) do not take DML locks, queries are allowed on a partition which is being SPLIT or MOVEd while the SPLIT or MOVE is being processed. However, the current segments are dropped at the end of the operation, and the space may be reused. An error is signalled if the space is reused.

While an ALTER TABLE MOVE PARTITION, ALTER TABLE SPLIT PARTITION, ALTER TABLE EXCHANGE PARTITION, or Direct Path Load Table Partition is in progress on a partition:

- You can move, split, exchange, or direct path load other partitions in the same table.
- You can run queries on the table.
- You can execute DML operations on the table provided that they do not write to that partition.
- You can rebuild any local index partition other than the ones that correspond to that partition.
- You cannot run any maintenance operation on the table or its indexes other than the ones listed above.

While a CREATE INDEX or ALTER INDEX REBUILD PARTITION or ALTER INDEX DROP/SPLIT PARTIITON applied to a Usable partition (for a global index) is in progress:

- You can run queries on the underlying table.
- You can create other indexes on the table, rebuild partitions in existing indexes, or drop or split usable partitions in existing indexes.
- You cannot execute any DML operation on the table or run any maintenance operation on the table or its indexes other than the ones listed above.

While an ALTER INDEX REBUILD PARTITION (for a local index) is in progress on a partition which corresponds to an underlying table partition:

- You can move, split, or direct path load any partition except the underlying table partition.
- You can run queries on the table.
- You can execute DML operations on the table provided that they do not write to the underlying table partition.
- You can rebuild other partitions in the index. You can also create other indexes on the table or rebuild partitions in other indexes.
- You cannot run any maintenance operation on the table or its indexes other than the ones listed above.

Some maintenance operations on a partition of a table cause the global indexes of the table or the index partitions to become Unusable. An example is ALTER TABLE MOVE PARTITION. The DBA has to run a script that includes global index rebuilds in addition to the partition maintenance operation. Consequently from a user point of view these operations serialize access to the entire table. Operations such as ALTER TABLE MOVE/SPLIT PARTITION make Unusable any non-partitioned global indexes as well as all partitions of partitioned global indexes.

Note that table partition operations which mark all partitions of global indexes also mark one partition of local index (the partition corresponding to the table partition being operated on) Unusable.

Similarly some partition maintenance operations require disabling Referential Integrity Constraints before the operation, and re-enabling them afterwards. An example is a ALTER TABLE DROP PARTITION of a non-empty partition. The DBA has to run a script that includes constraint re-enabling in addition to the partition maintenance operation. Consequently from a user point of view these operations serialize access to the entire table.

Queries and Partition Maintenance Operations

Queries whose execution starts before invocation of a partition maintenance operation, or before dictionary updates are done during a partition maintenance operation, correctly access via Consistent Read the data of the affected partitions as existing at query snapshot time. The behavior of such queries after dictionary updates have been done is unpredictable, in the sense that some of the data existing at snapshot time may be retrieved or errors may be returned.

Queries that use a partitioned index, and that start with some of the index partitions marked as Index Unusable, return an error when they actually access one of these partitions for the first time. This happens even if the partition has been made usable after query start.

Cursor Invalidation

Although many of the new DDL statements are partition-based, cursor invalidation is still table-based. This means that any DDL statement that modifies table T also invalidates all cursors that depend on T, even if the statement affects only one partition P of T and the cursors do not access partition P.

Recoverable and Unrecoverable Operations

All partition maintenance operations can be run in recoverable (LOGGING) mode. However, some operations support a NOLOGGING option:

- Parallel CREATE TABLE ... AS SELECT
- CREATE INDEX
- Direct Path SQL*Loader
- Direct-load INSERT

LOGGING is the default, except when the database is operating in NOARCHIVELOG mode. In that case, NOLOGGING is the default. DDL and utility statements that do not support the LOGGING / NOLOGGING option always run in recoverable mode (LOGGING).

Note: [NO]LOGGING is not an attribute of an operation but of a physical object. Hence, you cannot specify [NO]LOGGING in INSERT, but rather if you want to alter the logging mode of a table or index(es) involved in an INSERT, you need to issue ALTER TABLE / INDEX [NO]LOGGING before issuing the INSERT statement. For more information, see “Logging Mode” on page 21-5.

Managing Indexes

You can always rename, change the physical storage attributes, or rebuild a partition of a local or global index. Changing how an index is partitioned must be handled differently depending on whether the index is local or global.

Local Indexes

Oracle guarantees that the partitioning of a local index matches the partitioning of the underlying table. It does this by automatically creating or dropping index partitions as necessary when you alter the underlying table. You cannot explicitly add, drop, or split a partition in a local index.

For each local index:

- When you add a partition to the underlying table, Oracle automatically creates a new index partition with the same partition bound as the new table partition.
- When you drop a partition in the underlying table, Oracle automatically drops the corresponding index partition.
- When you split a partition in the underlying table, Oracle automatically splits the corresponding index partition. The two new index partitions have the same partition bounds as the new table partitions.

  Note that local index partitions produced as a result of splitting a parent table partition are marked Unusable if a corresponding table partition is non-empty.

When Oracle creates a new local index partition (via ADD or SPLIT):

- It tries to assign it the same name as the corresponding table partition. If that fails, it generates a name with the form SYS_P*nnn* (see “Partition Names” on page 9-14). You can rename the partition later.
- For ADD PARTITION, Oracle creates a segment with the default physical storage attributes of the base index. If a tablespace (other than DEFAULT, which causes local index partitions to be colocated with corresponding base table partitions) has been specified for the parent index, the index partition is placed in that tablespace. Otherwise, the tablespace in which the new index partition resides is that of the corresponding partition of the underlying table. You can modify these attributes later.
- For SPLIT PARTITION, attributes of the index partition being split are used for the index partitions resulting from the split. Partition names are the exception, although Oracle reuses table partition names when possible. For example, for a table partition with name TP and a local index with name IP, if TP is split into TP and TP1, then the names of the local index partitions are IP and TP1 (or a

system generated name if TP1 is already in use for that index). If TP is split into TP1 and TP2, then the local index partitions are TP1 and TP2. That is, if the table partition name is reused, Oracle tries to reuse the local index partition name also. All other attributes, however, are inherited from the index partition being split.

Global Indexes

The DBA is responsible for maintaining the partitioning of a global index. You can drop or split a partition in a global index. However, you cannot add a partition to a global index because the high partition of a global index always has a partition bound of MAXVALUE.

Rebuild Index Partition

The ALTER INDEX REBUILD PARTITION statement can be used to regenerate a single partition in a local or global partitioned index. This saves you from having to perform DROP INDEX and then CREATE INDEX, which would affect all partitions in the index.

ALTER INDEX REBUILD PARTITION has four important applications:

- To recluster an index partition to recover space and improve performance.
- To repair an index partition in case of a media failure on the volume where the index partition resides or a software corruption of the segment containing the index partition.
- To regenerate a local index partition after loading the underlying table partition with Import or SQL*Loader. These utilities offer a performance option to bypass index maintenance, mark the affected index partitions Index Unusable, and let the DBA rebuild them later. (Index Unusable is explained in the next section.) In other words, the strategy of “drop index then re-create index” can be replaced by a strategy of “mark index partition unusable then rebuild index partition.”
- To rebuild index partitions rendered unusable by partition maintenance operations on the underlying table.

Index Unusable Attribute

Some maintenance operations mark indexes *Index Unusable (IU)*. Index Unusable is an attribute of a non-partitioned index and of a partition in a partitioned index. When an index or index partition is marked IU, you get an error if you try to execute a SELECT or DML statement that requires the index (or partition).

When a single index partition is marked IU, you must rebuild the partition to make it valid again before using it. However, while one partition is marked IU the other partitions of the index are valid and you can execute SELECT or DML statements that require the index as long as the statements do not access the IU partition.

You can also split or rename the IU partition before rebuilding it, and you can drop an IU partition of a GLOBAL index.

When a non-partitioned index is marked IU, you can drop the indeYou can also drop an IU partition of a GLOBAL index.x and re-create it. You can also use ALTER INDEX REBUILD to rebuild a non-partitioned index.

Six types of maintenance operations can mark index partitions Index Unusable. In all cases, you must rebuild the index partitions when the operation is complete.

- Operations like Import Partition or conventional path SQL*Loader that offer an option to bypass local index maintenance. When the Import is complete, the affected local index partitions are marked IU.
- Direct path SQL*Loader leaves affected local index partitions and global indexes in an IU state if the index is out of date with respect to the data that it indexes. (Index Unusable was previously known as Direct Load State.) The index can be out of date for the following reasons:
  - The index could not be maintained by the load due to a space management error (for example, out of extents).
  - The user requested the SKIP_INDEX_MAINTENANCE option.
- Partition maintenance operations like ALTER TABLE MOVE PARTITION that change ROWIDs. These operations mark the affected local index partition and all global index partitions IU.
- Partition maintenance operations like ALTER TABLE TRUNCATE PARTITION or DROP PARTITION that remove rows from the table. These operations mark the affected local index partition and all global index partitions IU.
- Partition maintenance operations like ALTER TABLE SPLIT PARTITION that modify the partition definition of local indexes but do not automatically rebuild the index data to match the new definitions. These operations mark the affected local index partition(s) IU. (ALTER TABLE SPLIT PARTITION also marks all global index partitions IU because it results in changes to ROWIDs.)
- Index maintenance operations like ALTER INDEX SPLIT PARTITION that modify the partitioning definition of the index but do not automatically rebuild the affected partitions. These operations mark the affected index partition(s) IU. However, if you split a Usable partition of a global index, resulting partitions

are created usable. If the partition which was split was marked IU, then so are the partitions resulting from the split. (Note that dropping a partition of a global index which is either IU or is not empty causes the next partition of the index to become IU.)

Privileges for Partitioned Tables and Indexes

Privileges for partitions are granted on the parent table or index, not on individual partitions.

If a user or role has the privileges required to perform an Oracle operation on non-partitioned tables and indexes (including the necessary resource privileges), then the same Oracle operations are allowed on partitioned tables and indexes. For example:

- If you can create non-partitioned tables, then you can create partitioned tables.
- If you can drop non-partitioned indexes, then you can drop partitioned indexes.
- If you can add a column via ALTER to non-partitioned tables, then you can add a column via ALTER to partitioned tables.

If a user or role has the privileges required to perform an ALTER operation on a table or index, then the new ALTER operations on partitions of the table or index can be invoked, with these exceptions:

- The DROP ANY TABLE privilege (in addition to ALTER privilege) is required by a user that is not the table owner for:
  - ALTER TABLE DROP PARTITION
  - ALTER TABLE TRUNCATE PARTITION
- The following operations require space quota in the tablespace in which space is to be acquired, in addition to the ALTER privilege:
  - ALTER INDEX MODIFY PARTITION
  - ALTER INDEX REBUILD PARTITION
  - ALTER INDEX SPLIT PARTITION
  - ALTER TABLE ADD PARTITION
  - ALTER TABLE MODIFY PARTITION
  - ALTER TABLE MOVE PARTITION
  - ALTER TABLE SPLIT PARTITION

Auditing for Partitioned Tables and Indexes

All of the ALTER TABLE PARTITION operations are audited just like ALTER TABLE operations. No new audit attributes are used for partitions.

SQL Extension: Partition-Extended Table Name

Partition-level bulk operations are restricted to just the rows of a particular partition; for example, a user who wants to drop a partition without making all the global indexes UNUSABLE would want to delete all the rows from just that partition.

Such operations are very naturally expressed using the partition-extended table name syntax. Trying to phrase the same operation with a where-clause predicate becomes fairly cumbersome especially when the range partitioning key uses multiple columns.

The table specification syntax for the following DML statements may contain an optional partition specification for non-remote partitioned tables:

- INSERT
- UPDATE
- DELETE
- LOCK TABLE
- SELECT

For example:

```
SELECT * FROM schema.table PARTITION part_name;
```

This syntax provides a simple way of viewing individual partitions as tables: A view can be created which selects from just one partition using the partition-extended table name, and this view can be used in lieu of a table.

With such views you can also build partition-level access control mechanisms by granting (revoking) privileges on these views to (from) other users or roles. For application portability and ANSI syntax compliance you may use views to insulate your applications from this Oracle proprietary extension.

The use of partition-extended table names has the following restrictions:

1. *A partition-extended table name cannot refer to a remote schema object.*

   A partition-extended table name cannot contain a dblink or a synonym which translates to a table with a dblink. If you need to use remote partitions, you can

create a view at the remote site which uses the partition-extended table name syntax and refer to that remote view.

2. *The partition-extended table name syntax is not supported by PL/SQL.*

   A SQL statement using the partition-extended table name syntax cannot be used in a PL/SQL block, though it can be used through dynamic SQL via the DBMS_SQL package. Again, if you need to refer to a partition within a PL/SQL block you can instead use views which in turn use the partition-extended table name syntax.

3. *Only base tables are allowed.*

   A partition extension must be specified with a base table. No synonyms, views, or any other schema objects are allowed.

Examples of Partition-Extended Table Names

The following statements contain valid partition-extended table names:

```
SELECT * FROM sales PARTITION (nov95) s
  WHERE s.amount_of_sale > 1000;

UPDATE sales PARTITION (feb96) s
  SET s.account_name = UPPER(s.account_name);

DELETE FROM sales PARTITION (nov95)
  WHERE amount_of_sale != 0;

INSERT INTO sales PARTITION (oct95)
  SELECT * FROM lastest_data;

INSERT INTO sales PARTITION (oct95) VALUES (...);

INSERT INTO sales PARTITION (oct95) (acct_no, ..., week_no) VALUES (...);

LOCK TABLE sales PARTITION (jun95) IN EXCLUSIVE MODE;

CREATE VIEW sales_feb96 AS
  SELECT * FROM sales PARTITION (feb96);
```

10

Built-In 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 built-in datatypes, their properties, and how they map to non-Oracle datatypes. Topics include:

- Oracle Datatypes
  - Character Datatypes
  - NUMBER Datatype
  - DATE Datatype
  - LOB Datatypes
  - RAW and LONG RAW Datatypes
  - ROWID Datatype
  - MLSLABEL Datatype
- ANSI, DB2, and SQL/DS Datatypes
- Data Conversion

Oracle Datatypes

You can use the following built-in datatypes in column definitions:

- Character Datatypes
  - CHAR Datatype
  - VARCHAR2 Datatype
  - VARCHAR Datatype
  - NCHAR and NVARCHAR2 Datatypes
  - LONG Datatype
- NUMBER Datatype
- DATE Datatype
- LOB Datatypes
  - BLOB Datatype
  - CLOB and NCLOB Datatypes
  - BFILE Datatype
- RAW and LONG RAW Datatypes
- ROWID Datatype
- MLSLABEL Datatype

> **Additional Information:** PL/SQL has additional datatypes, such as BOOLEAN, reference types, composite types (collections and records), and user-defined subtypes. See the *PL/SQL User's Guide and Reference* for information about PL/SQL datatypes.

Character Datatypes

The character datatypes store character (alphanumeric) data in strings, with byte values corresponding to the character encoding scheme (generally called a character set or code page).

The database's character set is established when you create the database, and never changes. Examples of character sets are 7-bit ASCII (American Standard Code for Information Interchange), EBCDIC (Extended Binary Coded Decimal Interchange Code), Code Page 500, and Japan Extended UNIX. Oracle supports both single-byte and multibyte encoding schemes.

> **Additional Information:** See *Oracle8 Application Developer's Guide* for information about how to select a character datatype.

CHAR Datatype

The CHAR datatype stores **fixed**-length character strings. When you create a table with a CHAR column, you must specify a string length (in bytes, not characters) between 1 and 2000 for the CHAR column width. (The default is 1.) Oracle then guarantees that:

- When you insert or update a row in the table, the value for the CHAR column has the fixed length.
- If you give a shorter value, 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**.

> **Additional Information:** See *Oracle8 SQL Reference* for 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 string length (in bytes, not characters) between 1 and 4000 for the VARCHAR2 column. For each row, Oracle stores each value in the column as a variable-length field (unless a value exceeds the column's maximum length, in which case Oracle returns an error).

For example, assume you declare a column VARCHAR2 with a maximum size of 50 characters. In a single-byte character set, if only 10 characters are given for the VARCHAR2 column value in a particular row, the column in the row's row piece stores only the 10 characters (10 bytes), not 50.

Oracle compares VARCHAR2 values using the **nonpadded comparison semantics**.

> **Additional Information:** See *Oracle8 SQL Reference.*

VARCHAR Datatype

The VARCHAR datatype is currently synonymous with the VARCHAR2 datatype. However, in a future version of Oracle, VARCHARmight store variable-length char-

acter strings compared with different comparison semantics. Therefore, you should use the VARCHAR2 datatype to store variable-length character strings.

Column Lengths for Character Datatypes and NLS Character Sets

The Oracle National Language Support (NLS) feature allows the use of various character sets for the character datatypes. National Language Support enables you to process single-byte and multi-byte character data and convert between character sets. Client sessions can use national character sets different from the database character set.

You should consider the size of characters when you specify the column length for character datatypes. You must consider this issue when estimating space for tables with columns that contain character data.

> **Additional Information:** See *Oracle8 Reference* and *Oracle8 Utilities* for more information about the NLS feature of Oracle.

NCHAR and NVARCHAR2 Datatypes

The NCHAR and NVARCHAR2 datatypes store NLS character data. The NCHAR datatype stores fixed-length character strings that correspond to a fixed-length or variable-length national character set. The NVARCHAR2 datatype stores variable-length character strings.

When you create a table with an NCHAR or NVARCHAR2 column, you specify a maximum size that is either the number of characters (for a fixed-length national character set) or the number of bytes (for a variable-length national character set).

- The maximum length for an NCHAR column is 2000 bytes, or the number of characters that can be stored in 2000 bytes.
- The maximum length for an NVARCHAR2 column is 4000 bytes, or the number of characters that can be stored in 4000 bytes.

> **Additional Information:** See *Oracle8 Reference* and *PL/SQL User's Guide and Reference* for more information about NCHAR and NVARCHAR2 datatypes.

LOB Character Datatypes

The LOB datatypes for character data are CLOB and NCLOB. They can store up to four gigabytes of character data (CLOB) or national character set data (NCLOB). These datatypes are described in "LOB Datatypes" on page 10-9.

LONG Datatype

Columns defined as LONG can store variable-length character data containing up to two gigabytes of information. LONG data is text data that is to be appropriately converted when moving among different systems.

LONG datatype columns are used in the data dictionary to store the text of view definitions. You can use LONG columns in SELECT lists, SET clauses of UPDATE statements, and VALUES clauses of INSERT statements.

Note: The LONG datatype is provided for backward compatibility with existing applications. In new applications, you should use CLOB and NCLOB datatypes for large amounts of character data.

Additional Information: The LONG datatype has many restrictions — see *Oracle8 Application Developer's Guide.*

Also see "RAW and LONG RAW Datatypes" on page 10-11 for information about the LONG RAW datatype.

NUMBER Datatype

The NUMBER datatype stores fixed and floating-point numbers. Numbers of virtually any magnitude can be stored and are guaranteed portable among different systems operating Oracle, up to 38 digits of precision.

The following numbers can be stored in a NUMBER column:

- positive numbers in the range 1 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 by importing from an Oracle Version 5 database)

For numeric columns you can specify the column as:

```
column_name NUMBER
```

Optionally, you can also specify a *precision* (total number of digits) and *scale* (number of digits to the right of the decimal point):

```
column_name NUMBER (precision, scale)
```

If a precision is not specified, the column stores values as given. If no scale is specified, the scale is zero.

Oracle guarantees portability of numbers with a precision equal to or less than 38 digits. You can specify a scale and no precision:

```
column_name NUMBER (*, scale)
```

In this case, the precision is 38 and the specified scale is maintained.

When you specify numeric fields, it is a good idea to specify the precision and scale; this provides extra integrity checking on input.

Table 10–1 shows examples of how data would be stored using different scale factors.

Table 10–1 How Scale Factors Affect Numeric Data Storage

Input Data	Specified As	Stored As
7,456,123.89	NUMBER	7456123.89
7,456,123.89	NUMBER(*,1)	7456123.9
7,456,123.89	NUMBER(9)	7456124
7,456,123.89	NUMBER(9,2)	7456123.89
7,456,123.89	NUMBER(9,1)	7456123.9
7,456,123.89	NUMBER(6)	(not accepted, exceeds precision)
7,456,123.89	NUMBER(7,-2)	7456100

If you specify a negative scale, Oracle rounds the actual data to the specified number of places to the left of the decimal point. For example, specifying (7,-2) means Oracle should round to the nearest hundredths, as shown in Table 10–1.

For input and output of numbers, the standard Oracle default decimal character is a period, as in the number “1234.56”. (The decimal is the character that separates the integer and decimal parts of a number.) You can change the default decimal character with the initialization parameter NLS_NUMERIC_CHARACTERS. You can also change it for the duration of a session with the ALTER SESSION statement. To enter numbers that do not use the current default decimal character, use the TO_NUMBER function.

Internal Numeric Format

Oracle stores numeric data in variable-length format. Each value is stored in scientific notation, with one byte used to store the exponent and up to 20 bytes to store the mantissa. (The resulting value is limited to 38 digits of precision.) Oracle does not store leading and trailing zeros. For example, the number 412 is stored in a format similar to 4.12 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; positive infinity requires two bytes.

DATE Datatype

The DATE datatype stores point-in-time values (dates and times) in a table. The DATE datatype stores the year (including the century), the month, the day, the hours, the minutes, and the seconds (after midnight).

Oracle can store dates in the Julian era, ranging from January 1, 4712 BCE through December 31, 4712 CE (Common Era). Unless BCE ('BC' in the format mask) is specifically used, CE date entries are the default.

Oracle uses its own internal format to store dates. Date data is stored in fixed-length fields of seven bytes each, corresponding to century, year, month, day, hour, minute, and second.

For input and output of dates, the standard Oracle default date format is DD-MON-YY, as below:

```
'13-NOV-92'
```

You can change this default date format for an instance with the parameter NLS_DATE_FORMAT. You can also change it during a user session with the

ALTER SESSION statement. To enter dates that are not in standard Oracle date format, use the TO_DATE function with a format mask:

```
TO_DATE ('November 13, 1992', 'MONTH DD, YYYY')
```

> **Note:** If you use the standard date format DD-MON-YY, YY gives the year in the 20th century (for example, 31-DEC-92 is December 31, 1992). If you want to indicate years in any century other than the 20th century, use a different format mask, as shown above.

Oracle stores time in 24-hour format — HH:MI:SS. By default, the time in a date field is 00:00:00 A.M. (midnight) if no time portion is entered. In a time-only entry, the date portion defaults to the first day of the current month. To enter the time portion of a date, use the TO_DATE function with a format mask indicating the time portion, as in

```
INSERT INTO birthdays (bname, bday) VALUES
    ('ANDY',TO_DATE('13-AUG-66 12:56 A.M.','DD-MON-YY HH:MI A.M.'));
```

Using Julian Dates

Julian dates allow continuous dating by the number of days from a common reference. (The reference is 01-01-4712 years BCE, so current dates are somewhere in the 2.4 million range.) A Julian date is nominally a noninteger, the fractional part being a portion of a day. Oracle uses a simplified approach that results in integer values. Julian dates can be calculated and interpreted differently; the calculation method used by Oracle results in a seven-digit number (for dates most often used), such as 2449086 for 08-APR-93.

> **Note:** Oracle Julian dates might not be compatible with Julian dates generated by other date algorithms.

The format mask "J" can be used with date functions (TO_DATE or TO_CHAR) to convert date data into Julian dates. For example, the following query returns all dates in Julian date format:

```
SELECT TO_CHAR (hiredate, 'J') FROM emp;
```

You must use the TO_NUMBER function if you want to use Julian dates in calculations. You can use the TO_DATE function to enter Julian dates:

```
INSERT INTO emp (hiredate) VALUES (TO_DATE(2448921, 'J'));
```

Date Arithmetic

Oracle date arithmetic takes into account the anomalies of the calendars used throughout history. For example, the switch from the Julian to the Gregorian calendar, 15-10-1582, eliminated the previous 10 days (05-10-1582 through 14-10-1582). The year 0 does not exist.

You can enter missing dates into the database, but they are ignored in date arithmetic and treated as the next "real" date. For example, the next day after 04-10-1582 is 15-10-1582, and the day following 05-10-1582 is also 15-10-1582.

Note: This discussion of date arithmetic may not apply to all countries' date standards (such as those in Asia).

Centuries and the Year 2000

Oracle stores year data with the century information. For example, the Oracle database stores 1996 or 2001, and not just 96 or 01. The DATE datatype always stores a four-digit year internally, and all other dates stored internally in the database have four digit years. Oracle utilities such as import, export, and recovery also deal properly with four-digit years.

However, some applications might be written with an assumption about the year (such as assuming that everything is 19xx). Application programmers should therefore review and test their code with regard to the year 2000.

> **Additional Information:** For more information about centuries and date format masks, see *Oracle8 Application Developer's Guide*. For general information about date format codes, see *Oracle8 SQL Reference*.

LOB Datatypes

The LOB datatypes BLOB, CLOB, NCLOB, and BFILE enable you to store large blocks of unstructured data (such as text, graphic images, video clips, and sound waveforms) up to four gigabytes in size. They provide efficient, random, piece-wise access to the data.

LOB datatypes differ from LONG and LONG RAW datatypes in several ways. For example, LOB datatypes (except NCLOB) can be attributes of a user-defined object type but LONG datatypes cannot. The maximum size of a LOB is four gigabytes,

but the maximum size of a LONG is two gigabytes. LOBs support random access to data, but LONGs support only sequential access.

LOB datatypes can be stored inline (within a table), out-of-line (within a tablespace, using a LOB locator), or in an external file (BFILE datatypes).

> **Additional Information:** See *Oracle8 Application Developer's Guide* for more information about LOB storage and LOB locators.

You can use SQL statements to define LOB columns in a table and LOB attributes in a user-defined object type. When defining LOBs in a table, you can explicitly specify the tablespace and storage characteristics for each LOB.

See "Default Logging Mode" on page 21-7 for information about the LOB attribute LOGGING or NOLOGGING.

BLOB Datatype

The BLOB datatype stores unstructured binary data in the database. BLOBs can store up to four gigabytes of binary data.

BLOBs participate fully in transactions. Changes made to a BLOB value by the DBMS_LOB package, PL/SQL, or the OCI can be committed or rolled back. However, BLOB locators cannot span transactions or sessions.

CLOB and NCLOB Datatypes

The CLOB and NCLOB datatypes store up to four gigabytes of character data in the database. CLOBs store single-byte character set data and NCLOBs store fixed-length multibyte national character set data (NCHAR data).

CLOBs and NCLOBs participate fully in transactions. Changes made to a CLOB or NCLOB value by the DBMS_LOB package, PL/SQL, or the OCI can be committed or rolled back. However, CLOB and NCLOB locators cannot span transactions or sessions.

You cannot create an object type with NCLOB attributes, but you can specify NCLOB parameters in a method for an object type.

BFILE Datatype

The BFILE datatype stores unstructured binary data in operating-system files outside the database. A BFILE column or attribute stores a file locator that points to an external file containing the data. BFILEs can store up to four gigabytes of data.

BFILEs are read-only; you cannot modify them. They support only random (not sequential) reads, and they do not participate in transactions. The underlying oper-

ating system must maintain the file integrity and durability for BFILEs. The database administrator must ensure that the file exists and that Oracle processes have operating-system read permissions on the file.

RAW and LONG RAW Datatypes

> **Note:** The RAW and LONG RAW datatypes are provided for backward compatibility with existing applications. For new applications, you should use the BLOB and BFILE datatypes for large amounts of binary data.

The RAW and LONG RAW datatypes are used for data that is not to be interpreted (not converted when moving data between different systems) by Oracle. These datatypes are intended for binary data or byte strings. For example, LONG RAW can be used to store graphics, sound, documents, or arrays of binary data; the interpretation is dependent on the use.

RAW is a variable-length datatype like the VARCHAR2 character datatype, except that Net8 (which connects user sessions to the instance) and the Import and Export utilities do not perform character conversion when transmitting RAW or LONG RAW data. In contrast, Net8 and Import/Export automatically convert CHAR, VARCHAR2, and LONG data between the database character set and the user session character set (set by the NLS_LANGUAGE parameter of the ALTER SESSION 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.

> **Additional Information:** See *Oracle8 Application Developer's Guide* for information about additional restrictions on the LONG RAW datatype.

ROWID Datatype

Oracle uses an extended ROWID datatype to store the address of every row in the database. The extended ROWID efficiently identifies rows in partitioned tables and indexes as well as nonpartitioned tables and indexes. It supports tablespace-releative data block addresses. A restricted ROWID datatype is also available for backward compatibility with existing applications.

ROWIDs and the ROWID Datatype

Every row in a nonclustered table 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, for example:

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

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, and you cannot delete a ROWID value. Oracle uses the ROWIDs in the pseudocolumn ROWID internally for various operations as described in "How ROWIDs Are Used" on page 10-15.

You can reference ROWIDs in the pseudocolumn ROWID like other table columns (used in SELECT lists and WHERE clauses), but ROWIDs are not stored in the database, nor are they database data. However, you can create tables that contain columns having the ROWID datatype, although Oracle does not guarantee that the values of such columns are valid ROWIDs.

Extended ROWIDs

Extended ROWIDs use a base 64 encoding of the physical address for each row selected. For example, the following query

```
SELECT ROWID, ename FROM emp
    WHERE deptno = 20;
```

might return the following row information:

```
ROWID              ENAME
------------------ ----------
AAAAaoAATAAABrXAAA BORTINS
AAAAaoAATAAABrXAAE RUGGLES
AAAAaoAATAAABrXAAG CHEN
AAAAaoAATAAABrXAAN BLUMBERG
```

An extended ROWID has a four-piece format, OOOOOOFFFBBBBBBRRR:

- OOOOOO: The **data object number** identifies the database segment (AAAAao in the example). Schema objects in the same segment, such as a cluster of tables, have the same data object number.
- FFF: The **datafile** that contains the row (file AAT in the example). File numbers are unique within a database.
- BBBBBB: The **data block** that contains the row (block AAABrX in the example). Block numbers are relative to their datafile, **not** tablespace. Therefore, two rows with identical block numbers could reside in two different datafiles of the same tablespace.
- RRR: The **row** in the block.

You can retrieve the data object number from data dictionary views USER_OBJECTS, DBA_OBJECTS, and ALL_OBJECTS. For example, the following query returns the data object number for the EMP table in the SCOTT schema:

```
SELECT DATA_OBJECT_ID FROM DBA_OBJECTS
    WHERE OWNER = 'SCOTT' AND OBJECT_NAME = 'EMP';
```

You can also use the DBMS_ROWID package to extract information from an extended ROWID or to convert a ROWID from extended format to restricted format (or vice versa).

> **Additional Information:** See the *Oracle8 Application Developer's Guide* for information about the DBMS_ROWID package.

Restricted ROWIDs

Restricted ROWIDs use a binary representation of the physical address for each row selected. When queried using SQL*Plus, the binary representation is converted to a VARCHAR2/hexadecimal representation. The following query

```
SELECT ROWID, ename FROM emp
    WHERE deptno = 30;
```

might return the following row information:

```
ROWID              ENAME
------------------ ----------
00000DD5.0000.0001 KRISHNAN
00000DD5.0001.0001 ARBUCKLE
00000DD5.0002.0001 NGUYEN
```

As shown above, a restricted ROWID's VARCHAR2/hexadecimal representation is in a three-piece format, *block.row.file*:

- The **data block** that contains the row (block DD5 in the example). Block numbers are relative to their datafile, **not** tablespace. Therefore, two rows with identical block numbers could reside in two different datafiles of the same tablespace.
- The **row** in the block that contains the row (rows 0, 1, 2 in the example). Row numbers of a given block always start with 0.
- The **datafile** that contains the row (file 1 in the example). The first datafile of every database is always 1, and file numbers are unique within a database.

Examples of Using ROWIDs

You can use the function SUBSTR to break the data in a ROWID into its components. For example, you can use SUBSTR to break an extended ROWID into its four components (database object, file, block, and row):

```
SELECT ROWID,
       SUBSTR(ROWID,1,6) "OBJECT",
       SUBSTR(ROWID,7,3) "FIL",
       SUBSTR(ROWID,10,6) "BLOCK",
       SUBSTR(ROWID,16,3) "ROW"
       FROM products;

ROWID               OBJECT  FIL  BLOCK   ROW
------------------  ------  ---  ------  ----
AAAA8mAALAAAAQkAAA  AAAA8m  AAL  AAAAQk  AAA
AAAA8mAALAAAAQkAAF  AAAA8m  AAL  AAAAQk  AAF
AAAA8mAALAAAAQkAAI  AAAA8m  AAL  AAAAQk  AAI
```

Or you can use SUBSTR to break a restricted ROWID into its three components (block, row, and file):

```
SELECT ROWID, SUBSTR(ROWID,15,4) "FILE",
       SUBSTR(ROWID,1,8) "BLOCK",
       SUBSTR(ROWID,10,4) "ROW"
       FROM products;

ROWID              FILE  BLOCK     ROW
-----------------  ----  --------  ----
00000DD5.0000.0001  0001  00000DD5  0000
00000DD5.0001.0001  0001  00000DD5  0001
00000DD5.0002.0001  0001  00000DD5  0002
```

ROWIDs can be useful for revealing information about the physical storage of a table's data. For example, if you are interested in the physical location of a table's rows (such as for table striping), the following query of an extended ROWID tells how many datafiles contain rows of a given table:

```
SELECT COUNT(DISTINCT(SUBSTR(ROWID,7,3))) "FILES" FROM tablename;

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

> **Additional Information:** For more information on how to use ROWIDs, refer to the *Oracle8 SQL Reference*, the *PL/SQL User's Guide and Reference*, *Oracle8 Tuning*, and other books that document Oracle tools and utilities.

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 format of ROWIDs varies according to the characteristics of the non-Oracle system. Furthermore, no standard translation to VARCHAR2/hexadecimal format is available. Programs can still use the ROWID datatype; however, they must use a nonstandard translation to hexadecimal format of length up to 256 bytes.

> **Additional Information:** 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 functions:

- ROWIDs are the fastest means of accessing particular rows.
- ROWIDs can be used to see how a table is organized.
- ROWIDs are unique identifiers for rows in a given table.

Before you use ROWIDs in DML statements, they should be verified and guaranteed not to change; the intended rows should be locked so they cannot be deleted. Under some circumstances, requesting data with an invalid ROWID could cause a statement to fail.

You can also create tables with columns defined using the ROWID datatype. For example, you can define an exception table with a column of datatype ROWID to store the ROWIDs of rows in the database that violate integrity constraints. Columns defined using the ROWID datatype behave like other table columns; values can be updated, and so on. Each value in a column defined as datatype ROWID requires six bytes to store pertinent column data.

MLSLABEL Datatype

Trusted Oracle provides the MLSLABEL datatype, which stores Trusted Oracle's internal representation of labels generated by multilevel secure operating systems. Trusted Oracle uses labels to control database access.

You can define a column using the MLSLABEL datatype in Oracle8 for compatibility with Trusted Oracle applications, but the only valid value for the column in Oracle8 is NULL.

When you create a table in Trusted Oracle, a column called ROWLABEL is automatically appended to the table. This column contains a label of the MLSLABEL datatype for every row in the table.

> **Additional Information:** See your *Trusted Oracle* documentation for more information about the MLSLABEL datatype, the ROWLABEL column, and Trusted Oracle.

Summary of Oracle Datatype Information

Table 10–2 summarizes the important information about each Oracle datatype.

Table 10–2 Summary of Oracle Built-In Datatypes

Datatype	Description	Column Length and Default
CHAR (*size*)	Fixed-length character data of length *size* bytes.	Fixed for every row in the table (with trailing blanks); maximum size is 2000 bytes per row, default size is 1 byte per row. Consider the character set (one-byte or multibyte) before setting *size*.
VARCHAR2 (*size*)	Variable-length character data. A maximum *size* must be specified.	Variable for each row, up to 4000 bytes per row. Consider the character set (one-byte or multibyte) before setting *size*.
NCHAR(*size*)	Fixed-length character data of length *size* characters or bytes, depending on the national character set.	Fixed for every row in the table (with trailing blanks). Column *size* is the number of characters for a fixed-width national character set or the number of bytes for a varying-width national character set. Maximum *size* is determined by the number of bytes required to store one character, with an upper limit of 2000 bytes per row. Default is 1 character or 1 byte, depending on the character set.
NVARCHAR2 (*size*)	Variable-length character data of length *size* characters or bytes, depending on national character set. A maximum *size* must be specified.	Variable for each row. Column *size* is the number of characters for a fixed-width national character set or the number of bytes for a varying-width national character set. Maximum *size* is determined by the number of bytes required to store one character, with an upper limit of 4000 bytes per row. Default is 1 character or 1 byte, depending on the character set.
LONG	Variable-length character data.	Variable for each row in the table, up to 2^31 - 1 bytes, or 2 gigabytes, per row.
NUMBER (*p*, *s*)	Variable-length numeric data. Maximum precision *p* and / or scale *s* is 38.	Variable for each row. The maximum space required for a given column is 21 bytes per row.

Table 10–2 Summary of Oracle Built-In Datatypes (Cont.)

Datatype	Description	Column Length and Default
DATE	Fixed-length date and time data, ranging from January 1, 4712 BCE to December 31, 4712 CE ("A.D.")	Fixed at 7 bytes for each row in the table. Default format is a string (such as DD-MON-YY) specified by NLS_DATE_FORMAT parameter.
RAW (*size*)	Variable-length raw binary data. A maximum *size* must be specified.	Variable for each row in the table, up to 2000 bytes per row.
LONG RAW	Variable-length raw binary data.	Variable for each row in the table, up to 2^31 - 1 bytes, or 2 gigabytes, per row.
BLOB	Binary data.	Up to 2^32 - 1 bytes, or 4 gigabytes.
CLOB	Single-byte character data.	Up to 2^32 - 1 bytes, or 4 gigabytes.
NCLOB	Single-byte or fixed-length multibyte national character set (NCHAR) data.	Up to 2^32 - 1 bytes, or 4 gigabytes.
BFILE	Binary data stored in an external file.	Up to 2^32 - 1 bytes, or 4 gigabytes.
ROWID	Binary data representing row addresses.	Fixed at 10 bytes (extended ROWID) or 6 bytes (restricted ROWID) for each row in the table.
MLSLABEL	Trusted Oracle datatype.	See your *Trusted Oracle* documentation.

ANSI, DB2, and SQL/DS Datatypes

The ANSI datatype conversions to Oracle datatypes are shown in Table 10–3. The ANSI / ISO datatypes NUMERIC, DECIMAL, and DEC can specify only fixed-point numbers. For these datatypes, *s* (scale) defaults to 0.

Table 10–3 ANSI Datatype Conversions to Oracle Datatypes

ANSI SQL Datatype	Oracle Datatype
CHARACTER (n), CHAR (n)	CHAR (n)
NUMERIC (p,s), DECIMAL (p,s), DEC (p,s)	NUMBER (p,s)
INTEGER, INT, SMALLINT	NUMBER (38)
FLOAT (p)	FLOAT (p)
REAL	FLOAT (63)
DOUBLE PRECISION	FLOAT (126)
CHARACTER VARYING(n), CHAR VARYING(n)	VARCHAR2 (n)

The IBM products SQL / DS, and DB2 datatypes TIME, 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.

Table 10–4 shows the DB2 and SQL / DS conversions.

Table 10–4 SQL/DS, DB2 Datatype Conversions to Oracle Datatypes

DB2 or SQL/DS Datatype	Oracle Datatype
CHARACTER (n)	CHAR (n)
VARCHAR (n)	VARCHAR2 (n)
LONG VARCHAR	LONG
DECIMAL (p,s)	NUMBER (p,s)
INTEGER, SMALLINT	NUMBER (38)
FLOAT (p)	FLOAT (p)
DATE	DATE

Data Conversion

In some cases, Oracle supplies data of one datatype where it expects data of a different datatype. This is allowed when Oracle can automatically convert the data to the expected datatype using one of the following functions:

- TO_NUMBER()
- TO_CHAR()
- TO_DATE()
- CHARTOROWID()
- ROWIDTOCHAR()
- HEXTORAW()
- RAWTOHEX()

Additional Information: The rules for implicit datatype conversions are explained in the *Oracle8 Application Developer's Guide*.

If you are using Trusted Oracle, see your *Trusted Oracle* documentation for additional information involving data conversions and the MLSLABEL datatype.

11

User-Defined Datatypes (Objects Option)

We must learn to explore all the options and possibilities that confront us in a complex and rapidly changing world.

James William Fulbright, *Speech in the Senate (1964)*

The Objects option allows users to define datatypes that model the structure and behavior of the data in their applications.

This chapter contains the following major sections:

- Introduction
- User-Defined Datatypes
- Application Interfaces

Attention: The features described in this chapter are available only if you have purchased Oracle8 Enterprise Edition with the Objects Option. Wherever the term Oracle server appears in this chapter it refers to the Oracle8 Enterprise Edition with the Objects Option. See *Getting to Know Oracle8 and the Oracle8 Enterprise Edition* for information about the features and options available with Oracle8 Enterprise Edition.

Introduction

Relational database management systems (RDBMSs) are the standard tool for managing business data. They provide fast, efficient, and completely reliable access to huge amounts of data for millions of businesses around the world every day.

The Objects option makes Oracle an object-relational database management system (ORDBMS), which means that users can define additional kinds of data — specifying both the structure of the data and the ways of operating on it — and use these types within the relational model. This approach adds value to the data stored in a database.

Oracle with the Objects option stores structured business data in its natural form and allows applications to retrieve it that way. For that reason it works efficiently with applications developed using object-oriented programming techniques.

Oracle's support for user-defined datatypes makes it easier for application developers to work with complex data like images, audio, and video.

Complex Data Models

The Oracle server allows you to define complex business models in SQL and make them part of your database schema. Applications that manage and share your data need only contain the application logic, not the data logic.

An Example

For example, your firm may use purchase orders to organize its purchasing, accounts payable, shipping, and accounts receivable functions.

A purchase order contains an associated supplier or customer and an indefinite number of line items. In addition, applications often need dynamically computed status information about purchase orders. For example, you may need the current value of the shipped or unshipped line items.

Later sections of this chapter show how you can define a schema object, called an *object type*, that serves as a template for all purchase order data in your applications. An object type specifies the elements, called *attributes*, that make up a structured data unit like a purchase order. Some attributes, such as the list of line items, may be other structured data units. The object type also specifies the operations, called *methods*, you can perform on the data unit, such as determining the total value of a purchase order.

You can create purchase orders that match the template and store them in table columns, just as you would numbers or dates.

You can also store purchase orders in *object tables*, where each row of the table corresponds to a single purchase order and the table columns are the purchase order's attributes.

Since the logic of the purchase order's structure and behavior is in your schema, your applications don't need to know the details and don't have to keep up with most changes.

Oracle uses schema information about object types to achieve substantial transmission efficiencies. A client-side application can request a purchase order from the server and receive all the relevant data in a single transmission. The application can then, without knowing storage locations or implementation details, navigate among related data items without further transmissions from the server.

Multimedia Datatypes

Many efficiencies of database systems arise from their optimized management of basic datatypes like numbers, dates, and characters. Facilities exist for comparing values, determining their distributions, building efficient indexes, and performing other optimizations.

Text, video, sound, graphics, and spatial data are examples of important business entities that don't fit neatly into those basic types. Oracle with the Objects option supports modeling and implementation of these complex datatypes.

User-Defined Datatypes

Chapter 10, "Built-In Datatypes" describes Oracle's built-in datatypes. The Objects option adds two categories of user-defined datatypes:

- object types
- collection types

User-defined datatypes use the built-in datatypes and other user-defined datatypes as the building blocks for datatypes that model the structure and behavior of data in applications.

User-defined types are schema objects. Their use is subject to the same kinds of administrative control as other schema objects (see Chapter 12, "Using User-Defined Datatypes").

Object Types

Object types are abstractions of the real-world entities — for example, purchase orders — that application programs deal with. An object type is a schema object with three kinds of components:

- A *name*, which serves to identify the object type uniquely within that schema.
- *Attributes*, which model the structure and state of the real world entity. Attributes are built-in types or other user-defined types.
- *Methods*, which are functions or procedures written in PL/SQL and stored in the database, or written in a language like C and stored externally. Methods implement operations the application can perform on the real world entity.

An object type is a template. A structured data unit that matches the template is called an *object*.

Purchase Order Example

Here is an example of how you might define object types called `external_person`, `lineitem`, and `purchase_order`.

The object types `external_person` and `lineitem` have attributes of built-in types. The object type `purchase_order` has a more complex structure, which closely matches the structure of real purchase orders.

The attributes of `purchase_order` are `id`, `contact`, and `lineitems`. The attribute `contact` is an object, and the attribute `lineitems` is a nested table (see "Nested Tables" on page 11-10).

```
CREATE TYPE external_person AS OBJECT (
  name        VARCHAR2(30),
  phone       VARCHAR2(20) );

CREATE TYPE lineitem AS OBJECT (
  item_name   VARCHAR2(30),
  quantity    NUMBER,
  unit_price  NUMBER(12,2) );

CREATE TYPE lineitem_table AS TABLE OF lineitem;

CREATE TYPE purchase_order AS OBJECT (
  id          NUMBER,
  contact     external_person,
  lineitems   lineitem_table,
```

```
  MEMBER FUNCTION
  get_value   RETURN NUMBER );
```

This is a simplified example. It does not show how to specify the body of the method `get_value`. Nor does it show the full complexity of a real purchase order.

> **Additional Information:** See *Oracle8 Application Developer's Guide* for a complete purchase order example.

An object type is a template. Defining it doesn't result in storage allocation. You can use `lineitem`, `external_person`, or `purchase_order` in SQL statements in most of the same places you can use types like NUMBER or VARCHAR2.

For example, you might define a relational table to keep track of your contacts:

```
CREATE TABLE contacts (
  contact     external_person
  date        DATE );
```

The `contact` table is a relational table with an object type defining one of its columns. Objects that occupy columns of relational tables are called *column objects* (see "Row Objects and Column Objects" on page 11-8).

Methods

In the example, `purchase_order` has a method named get_value. Each purchase order object has its own get_value method. For example, if `x` and `y` are PL/SQL variables that hold purchase order objects and `w` and `z` are variables that hold numbers, the following two statements can leave `w` and `z` with different values:

```
w = x.get_value();
z = y.get_value();
```

After those statements, `w` has the value of the purchase order referred to by variable `x`; `z` has the value of the purchase order referred to by variable `y`.

The term `x.get_value` () is an invocation of the method `get_value`. Method definitions can include parameters, but `get_value` does not need them, because it finds all of its arguments among the attributes of the object to which its invocation is tied. That is, in the first of the sample statements, it computes its value using the attributes of purchase order `x`. In the second it computes its value using the attributes of purchase order `y`. This is called the *selfish style* of method invocation.

Every object type also has one implicitly defined method that is not tied to specific objects, the object type's constructor method.

Object Type Constructor Methods Every object type has a system-defined *constructor method*, that is, a method that makes a new object according to the object type's specification. The name of the constructor method is the name of the object type. Its parameters have the names and types of the object type's attributes. The constructor method is a function. It returns the new object as its value.

For example, the expression

```
purchase_order(
  1000376,
  external_person ("John Smith","1-800-555-1212"),
  NULL )
```

represents a purchase order object with the following attributes:

```
id         1000376
contact    external_person("John Smith","1-800-555-1212")
lineitems  NULL
```

The expression `external_person ("John Smith", "1-800-555-1212")` is an invocation of the constructor function for the object type `external_person`. The object that it returns becomes the contact attribute of the purchase order.

See "Nulls" on page 12-6 for a discussion of null objects and null attributes.

Comparison Methods Methods play a role in comparing objects. Oracle has facilities for comparing two data items of a given built-in type (for example, two numbers), and determining whether one is greater than, equal to, or less than the other. Oracle cannot, however, compare two items of an arbitrary user-defined type without further guidance from the definer. Oracle provides two ways to define an order relationship among objects of a given object type: *map* methods and *order* methods.

Map methods use Oracle's ability to compare built-in types. Suppose, for example, that you have defined an object type called `rectangle`, with attributes `height` and `width`. You can define a map method area that returns a number, namely the product of the rectangle's `height` and `width` attributes. Oracle can then compare two rectangles by comparing their areas.

Order methods are more general. An order method uses its own internal logic to compare two objects of a given object type. It returns a value that encodes the order relationship. For example, it may return -1 if the first is smaller, 0 if they are equal, and 1 if the first is larger.

Suppose, for example, that you have defined an object type called `address`, with attributes `street`, `city`, `state`, and `zip`. The terms "greater than" and "less

than" may have no meaning for addresses in your application, but you may need to perform complex computations to determine when two addresses are equal.

In defining an object type, you can specify either a map method or an order method for it, but not both. If an object type has no comparison method, Oracle cannot determine a greater than or less than relationship between two objects of that type. It can, however, attempt to determine whether two objects of the type are equal.

Oracle compares two objects of a type that lacks a comparison method by comparing corresponding attributes:

- If all the attributes are non-null and equal, Oracle reports that the objects are equal.
- If there is an attribute for which the two objects have unequal non-null values, Oracle reports them unequal.
- Otherwise, Oracle reports that the comparison is not available (null).

> **Additional Information:** For examples of how to specify and use comparison methods, see *Oracle8 Application Developer's Guide*.

Object Tables

An *object table* is a special kind of table that holds objects and provides a relational view of the attributes of those objects.

For example, the following statement defines an object table for objects of the `external_person` type defined earlier:

```
CREATE TABLE external_person_t OF external_person;
```

Oracle allows you to view this table in two ways:

- A single column table in which each entry is an `external_person` object.
- A multi-column table in which each of the attributes of the object type `external_person`, namely `name` and `phone`, occupies a column.

For example, you can execute the following instructions:

```
INSERT INTO external_person_t VALUES (
       "John Smith",
       "1-800-555-1212" );

SELECT VALUE(p) FROM external_person_t p
        WHERE p.name = "John Smith";
```

The first instruction inserts a purchase order object into `external_person_t` as a multi-column table. The second selects from `external_person_t` as a single column table.

Row Objects and Column Objects Objects that appear in object tables are called *row objects*. Objects that appear only in table columns or as attributes of other objects are called *column objects*.

REFs

In the relational model, foreign keys express many-to-one relationships. Oracle with the Objects option provides a more efficient means of expressing many-to-one relationships when the "one" side of the relationship is a row object. Oracle gives every row object a unique, immutable identifier, called an *object identifier*. Oracle provides no documentation of or access to the internal structure of object identifiers. This structure can change at any time.

An object identifier allows the corresponding row object to be referred to from other objects or from relational tables. A built-in datatype called REF represents such references. A REF encapsulates a reference to a row object of a specified object type.

An object view (see Chapter 13, "Object Views") is a virtual object table. Its rows are row objects. Oracle materializes object identifiers, which it does not store persistently, from primary keys in the underlying table or view. Oracle uses these identifiers to construct REFs to the row objects in the object view.

You can use a REF to examine or update the object it refers to. You can also use a REF to obtain a copy of the object it refers to. The only changes you can make to a REF are to replace its contents with a reference to a different object of the same object type or to assign it a null value.

Scoped REFs In declaring a column type, collection element, or object type attribute to be a REF, you can constrain it to contain only references to a specified object table. Such a REF is called a *scoped REF*. Scoped REFs require less storage space and allow more efficient access than unscoped REFs.

Dangling REFs It is possible for the object identified by a REF to become unavailable — through either deletion of the object or a change in privileges. Such a REF is called *dangling*. Oracle SQL provides a predicate (called IS DANGLING) to allow testing REFs for this condition.

Dereferencing REFs Accessing the object referred to by a REF is called *dereferencing* the REF. Oracle provides the DEREF operator to do this. Dereferencing a dangling REF results in a null object.

Oracle provides *implicit dereferencing* of REFs. For example, consider the following:

```
CREATE TYPE person AS OBJECT (
  name    VARCHAR2(30),
  manager REF person );
```

If x represents an object of type `person`, then the expression

```
x.manager.name
```

represents a string containing the `name` attribute of the `person` object referred to by the `manager` attribute of `x`. The above expression is a shortened form of:

```
y.name, where y = DEREF(x.manager)
```

Obtaining REFs You can obtain a REF to a row object by selecting the object from its object table and applying the REF operator. For example, you can obtain a REF to the purchase order with identification number 1000376 as follows:

```
DECLARE OrderRef REF to purchase_order;

SELECT REF(po) INTO OrderRef
               FROM purchase_order_table po
               WHERE po.id = 1000376;
```

For more on storage of objects and REFs, see "Storage of User-Defined Types" on page 12-4.

> **Additional Information:** For examples of how to use REFs, see *Oracle8 Application Developer's Guide*.

Collection Types

Each collection type describes a data unit made up of an indefinite number of elements, all of the same datatype. The collection types are *array types* and *table types*.

Array types and table types are schema objects. The corresponding data units are called *VARRAYs* and *nested tables*. When there is no danger of confusion, we often refer to the collection types as VARRAYs and nested tables.

Collection types have constructor methods. The name of the constructor method is the name of the type, and its argument is a comma-separated list of the new collec-

tion's elements. The constructor method is a function. It returns the new collection as its value.

An expression consisting of the type name followed by empty parentheses represents a call to the constructor method to create an empty collection of that type. An empty collection is different from a null.

VARRAYs

An *array* is an ordered set of data *elements*. All elements of a given array are of the same datatype. Each element has an *index*, which is a number corresponding to the element's position in the array.

The number of elements in an array is the *size* of the array. Oracle allows arrays to be of variable size, which is why they are called VARRAYs. You must specify a maximum size when you declare the array type.

For example, the following statement declares an array type:

```
CREATE TYPE prices AS VARRAY(10) OF NUMBER(12,2);
```

The VARRAYs of type `prices` have no more than ten elements, each of datatype NUMBER(12,2).

Creating an array type does not allocate space. It defines a datatype, which you can use as

- The datatype of a column of a relational table.
- An object type attribute
- A PL/SQL variable, parameter, or function return type.

A VARRAY is normally stored in line, that is, in the same tablespace as the other data in its row. If it is sufficiently large, however, Oracle stores it as a BLOB (see "Import/Export of User-Defined Types" on page 12-15).

> **Additional Information:** For more information on using VARRAYs, see *Oracle8 Application Developer's Guide*.

Nested Tables

A *nested table* is an unordered set of data *elements*, all of the same datatype. It has a single column, and the type of that column is a built-in type or an object type. If an object type, the table can also be viewed as a multi-column table, with a column for each attribute of the object type.

For example, in the purchase order example, the following statement declares the table type used for the nested tables of line items:

```
CREATE TYPE lineitem_table AS TABLE OF lineitem;
```

A table type definition does not allocate space. It defines a type, which you can use as

- The datatype of a column of a relational table.
- An object type attribute.
- A PL/SQL variable, parameter, or function return type.

When a table type appears as the type of a column in a relational table or as an attribute of the underlying object type of an object table, Oracle stores all of the nested table data in a single table, which it associates with the enclosing relational or object table (see "Nested Tables" on page 12-5). For example, the following statement defines an object table for the object type `purchase_order`:

```
CREATE TABLE purchase_order_table OF purchase_order
   NESTED TABLE lineitems STORE AS lineitems_table;
```

The second line specifies `lineitems_table` as the storage table for the `lineitems` attributes of all of the `purchase_order` objects in `purchase_order_table`.

A convenient way to access the elements of a nested table individually is to use a nested cursor.

> **Additional Information:** See *Oracle8 Reference* for information about nested cursors, and see *Oracle8 Application Developer's Guide* for more information on using nested tables.

Application Interfaces

Oracle provides a number of facilities for using user-defined datatypes in application programs:

- SQL
- PL/SQL
- Pro*C/C++
- OCI
- OTT

SQL

Oracle SQL DDL provides the following support for user-defined datatypes:

- defining object types, nested tables, and arrays
- specifying privileges
- specifying table columns of user-defined types
- creating object tables

Oracle SQL DML provides the following support for user-defined datatypes:

- querying and updating objects and collections
- manipulating REFs

> **Additional Information:** For a complete description of Oracle SQL syntax, see *Oracle8 SQL Reference*.

PL/SQL

PL/SQL is a procedural language that extends SQL. It offers modern software engineering features like packages, data encapsulation, information hiding, overloading, and exception handling. It is the language used for stored procedures.

PL/SQL allows use from within functions and procedures of the SQL features that support user-defined types.

The parameters and variables of PL/SQL functions and procedures can be of user-defined types.

PL/SQL provides all the capabilities necessary to implement the methods associated with object types. These methods (functions and procedures) reside on the server as part of a user's schema.

> **Additional Information:** For a complete description of PL/SQL, see *PL/SQL User's Guide and Reference*.

Pro*C/C++

The Oracle Pro*C/C++ precompiler allows programmers to use user-defined datatypes in C and C++ programs.

Pro*C developers can use the Object Type Translator to map Oracle object types and collections into C datatypes to be used in the Pro*C application.

Pro*C provides compile time type checking of object types and collections and automatic type conversion from database types to C datatypes.

Pro*C includes an EXEC SQL syntax to create and destroy objects and offers two ways to access objects in the server:

- SQL statements and PL/SQL functions or procedures embedded in Pro*C programs.
- A simple interface to the object cache (described under OCI), where objects can be accessed by traversing pointers, then modified and updated on the server.

> **Additional Information:** For a complete description of the Pro*C precompiler, see *Pro*C/C++ Precompiler Programmer's Guide*.

OCI

The Oracle call interface (OCI) is a set of C language interfaces to the Oracle server. It provides programmers great flexibility in using the server's capabilities.

An important component of OCI is a set of calls to allow application programs to use a workspace called the object cache. The *object cache* is a memory block on the client side that allows programs to store entire objects and to navigate among them without round trips to the server.

The object cache is completely under the control and management of the application programs using it. The Oracle server has no access to it. The application programs using it must maintain data coherency with the server and protect the workspace against simultaneous conflicting access.

OCI provides functions to

- Access objects on the server using SQL.
- Access, manipulate and manage objects in the object cache by traversing pointers or REFs.
- Convert Oracle dates, strings and numbers to C data types.
- Manage the size of the object cache's memory.

OCI improves concurrency by allowing individual objects to be locked. It improves performance by supporting complex object retrieval.

OCI developers can use the object type translator to generate the C datatypes corresponding to a Oracle object types.

Additional Information: For a complete description of OCI, see *Oracle Call Interface Programmer's Guide*.

OTT

The Oracle type translator (OTT) is a program that automatically generates C language structure declarations corresponding to object types. OTT facilitates using the Pro*C precompiler and the OCI server access package.

Additional Information: For complete information about OTT, see *Oracle Call Interface Programmer's Guide* and *Pro*C/C++ Precompiler Programmer's Guide*.

12

Using User-Defined Datatypes

It is not enough to have a good mind. The main thing is to use it well.

René Descartes, *Le Discours de la Méthode*

This chapter covers the main concepts you need to understand to use user-defined datatypes. It contains the following major sections:

- References and Name Resolution
- Storage of User-Defined Types
- Properties of Object Attributes
- Privileges on User-Defined Types and Their Methods
- Dependencies and Incomplete Types
- Import/Export of User-Defined Types

Attention: The features described in this chapter are available only if you have purchased Oracle8 Enterprise Edition with the Objects Option.

See *Getting to Know Oracle8 and the Oracle8 Enterprise Edition* for information about the features and options available with Oracle8 Enterprise Edition.

References and Name Resolution

Oracle SQL is designed to be easy to use. For example, if `projects` is a table with a column called `assignment`, and `depts` is a table that does not contain a column called `assignment`, you can write

```
SELECT *
FROM projects
WHERE EXISTS
  (SELECT * FROM  depts
            WHERE assignment = task);
```

Oracle determines which table each column belongs to. You can, but don't have to, qualify the column names with table names:

```
SELECT *
FROM projects
WHERE EXISTS
  (SELECT * FROM  depts
            WHERE projects.assignment = depts.task);
```

You can, but don't have to, qualify the column names with table aliases:

```
SELECT *
FROM projects pj
WHERE EXISTS
  (SELECT * FROM  depts dp
            WHERE pj.assignment = dp.task);
```

Table Aliases

The first form of the SELECT statement above is the easiest to write and understand, but it can lead to undesired results if you later add an `assignment` column to the `depts` table and forget to change the query. Oracle automatically recompiles the query and the new version uses the `assignment` column from the `depts` table. This situation is called *inner capture*.

In order to avoid inner capture and similar misinterpretations of the intended meanings of SQL statements, Oracle requires you to use table aliases to qualify references to methods or attributes of objects. This also applies to attribute references via REFs. This requirement is called the *capture avoidance rule*.

For example, consider the following statements:

```
CREATE TYPE person AS OBJECT (ssno VARCHAR(20));
CREATE TABLE ptab1 OF person;
CREATE TABLE ptab2 (c1 person);
```

These define an object type `person` and two tables. The first is an object table for objects of type `person`. The second has a single column of type `person`.

Now consider the following queries:

```
SELECT      ssno FROM ptab1   ;  --Correct
SELECT   c1.ssno FROM ptab2   ;  --Wrong
SELECT p.c1.ssno FROM ptab2 p ;  --Correct
```

- In the first SELECT statement, `ssno` is the name of a column of `ptab1`. No further qualification is required.
- In the second SELECT statement, `ssno` is the name of an attribute of the `person` object in the column named `c1`. This reference requires a table alias.
- The third SELECT statement is the same as the second, but contains the required table alias, `p`.

Qualifying references to object attributes with table names rather than table aliases, even if the table names are further qualified by schema names, does not satisfy this requirement.

For example, you cannot, in a query, use the expression

```
scott.projects.assignment.duedate
```

to refer to the `duedate` attribute of the `assignment` column of the `projects` table of the `scott` schema.

Table aliases should be unique throughout a query and should not be the same as schema names that could legally appear in the query.

Note: Oracle recommends that you define table aliases in all UPDATE, DELETE, and SELECT statements and subqueries and use them to qualify column references, whether or not the columns contain object types.

Method Calls without Arguments

Methods are functions or subroutines. The proper syntax for invoking them uses parentheses following the method name to enclose any calling arguments. In order to avoid ambiguities, Oracle requires empty parentheses for method calls that do not have arguments.

For example, if `tb` is a table with column `c` of object type `t`, and `t` has a method `m` that does not take arguments, the following query illustrates the correct syntax:

```
SELECT p.c.m() FROM tb p;
```

This differs from the rules for PL/SQL functions and procedures, where the parentheses are optional for calls that have no arguments.

Storage of User-Defined Types

Oracle stores and manages data of user-defined types in tables. It automatically and invisibly maps the complex structure of user-defined types into the simple rectangular structure of tables.

Leaf-Level Attributes

The structure of an object type is like a tree. The branches that grow from the trunk go to the attributes. If an attribute is of an object type, that branch sprouts subbranches for the attributes of the new object type.

Ultimately each branch comes to an end at an attribute that is of a built-in type or a collection type. These are called *leaf-level attributes* of the original object type. Oracle provides a table column for each leaf-level attribute.

The leaf-level attributes that not collection types are called the *leaf-level scalar attributes* of the object type.

Row Objects

In an object table, every leaf-level scalar or REF attribute has a column in which Oracle stores its actual data. This is also true of VARRAYs, unless they are too large (see "VARRAYs" on page 12-5). Oracle stores leaf-level attributes of table types in separate tables associated with the object table. You must declare these tables as part of the object table declaration (see "Nested Tables" on page 12-5).

Access to individual attributes of objects in an object table is simply access to columns of the table. Accessing the value of the object itself causes Oracle to invoke the default constructor for the type, using the columns of the object table as arguments. That is, Oracle supplies a copy of the object.

Oracle stores the system-generated object identifier in a hidden column. Oracle uses the object identifier to construct REFs to the object.

Column Objects

When a table is defined with a column of an object type, Oracle invisibly adds columns to the table for the object type's leaf-level attributes. An additional column stores the NULL information of the object (that is, the atomic nulls of the top-level and the nested objects).

REFs

Oracle constructs a REF to a row object by invoking the built-in function REF on the row object. The constructed REF is made up of the object identifier, some metadata of the object table, and, optionally, the ROWID. An unscoped REF with ROWID to an object in an object table is 46 bytes in size. REFs to object views, REFs without ROWID, and scoped REFs are smaller.

The ROWID in a REF is used as a hint for efficient access. When Oracle dereferences a REF item, it uses the ROWID to choose a row; if the object identifier of the identified row matches the one in the REF, the access is successful. Otherwise, Oracle uses the index on the object identifier to identify the correct row.

The size of a REF in a column of REF type depends on the storage properties associated with the column. For example, if the column is declared as a REF WITH ROWID, Oracle stores the ROWID in the REF column; otherwise, it discards the ROWID.

If column is declared as REF with a SCOPE clause, then Oracle does not store the object table metadata and the ROWID in the column. A scoped REF is 16 bytes long.

Nested Tables

The rows of a nested table are stored in a separate storage table. You must supply a storage tablename when you define the table containing the nested table. If the table definition contains more than one table type — either in columns or in object types that appear in column definitions — you must supply a separate storage table for each.

For each nested table in the table definition, the associated storage table contains the rows of all instances of the given nested table in the rows of the parent table.

VARRAYs

All the elements of a VARRAY are stored in a single column. If the size of the array is smaller than 4000 bytes, Oracle stores it in line; if it is greater than 4000 bytes, Oracle stores it in a BLOB.

Properties of Object Attributes

Oracle allows you to specify some properties of object attributes:

- Nulls
- Defaults
- Constraints
- Indexes
- Triggers

Nulls

One possible property of a table column, object, object attribute, collection, or collection element is that it can be null. This means that the item has been initialized to NULL or has been left uninitialized. Usually this means that the value of the item is not yet known but might become available later.

An object whose value is NULL is called *atomically null*. In addition, attributes of an object can be null. These two uses of nulls are different.

For example, consider the `contacts` table defined as follows:

```
CREATE TYPE external_person AS OBJECT (
  name        VARCHAR2(30),
  phone       VARCHAR2(20) );

CREATE TABLE contacts (
  contact     external_person
  date        DATE );
```

The statement

```
INSERT INTO contacts VALUES (
  external_person (NULL, NULL),
 '24 Jun 1997' );
```

gives a different result from

```
INSERT INTO contacts VALUES (
  NULL,
 '24 Jun 1997' );
```

In both cases, Oracle allocates space in `contacts` for a new row and sets its `date` column to the value given. In the first case, Oracle allocates space for an object in

the external_person column and sets each of its attributes to NULL. In the second case, it sets the external_person column to NULL and does not allocate space for an object.

A table row cannot be null. Therefore, Oracle does not allow you to set a row object to NULL. Similarly, a nested table of objects cannot contain an element whose value is NULL.

A nested table or array can be null. A null collection is different from an empty one, that is, a collection containing no elements.

Defaults

When you declare a table column to be of an object type or collection type, you can include a DEFAULT clause. This provides a value to use in cases where you do not explicitly specify a value for the column. The default clause must contain a *literal invocation* of the constructor method for that object or collection.

A *literal invocation* of a constructor method is defined recursively to be an invocation of the constructor method in which any arguments are either literals or literal invocations of constructor methods.

For example, consider the following statements:

```
CREATE TYPE person AS OBJECT (
  id        NUMBER
  name      VARCHAR2(30),
  address   VARCHAR2(30) );

CREATE TYPE people AS TABLE OF person;
```

The following is a literal invocation of the constructor method for the nested table type people:

```
people ( person(1, 'John Smith', '5 Cherry Lane'),
         person(2, 'Diane Smith', NULL) )
```

The following example shows how to use literal invocations of constructor methods to specify defaults:

```
CREATE TABLE department (
  d_no    CHAR(5) PRIMARY KEY,
  d_name  CHAR(20),
  d_mgr   person DEFAULT person(1,'John Doe',NULL),
  d_emps  people DEFAULT people() )
  NESTED TABLE d_emps STORE AS d_emps_tab;
```

Note that the term `people( )` is a literal invocation of the constructor method for an empty `people` table.

Constraints

You can define constraints on an object table just as you can on other tables.

You can define constraints on the leaf-level scalar attributes of a column object, with the exception of REFs that are not scoped (see "Scoped REFs" on page 11-8).

The following examples illustrate the possibilities.

The first example places a primary key constraint on the `ssno` column of the object table `person_extent`:

```
CREATE TYPE location (
  building_no NUMBER,
  city        VARCHAR2(40) );

CREATE TYPE person (
  ssno        NUMBER,
  name        VARCHAR2(100),
  address     VARCHAR2(100),
  office      location );

CREATE TABLE person_extent OF person (
  ssno        PRIMARY KEY );
```

The `department` table in the next example has a column whose type is the object type `location` defined in the previous example. The example defines constraints on scalar attributes of the `location` objects that appear in the `dept_loc` column of the table.

```
CREATE TABLE department (
  deptno      CHAR(5) PRIMARY KEY,
  dept_name   CHAR(20),
  dept_mgr    person,
  dept_loc    location,
  CONSTRAINT  dept_loc_cons1
      UNIQUE (dept_loc.building_no, dept_loc.city),
  CONSTRAINT  dept_loc_cons2
       CHECK (dept_loc.city IS NOT NULL) );
```

Indexes

You can define indexes on an object table or on the storage table for a nested table column or attribute just as you can on other tables.

You can define indexes on leaf-level scalar attributes of column objects, except that you can only define indexes on REF attributes or columns if the REF is scoped (see "Scoped REFs" on page 11-8).

The following example defines an index on an attribute of an object column:

```
CREATE TABLE department (
  deptno      CHAR(5) PRIMARY KEY,
  dept_name   CHAR(20),
  dept_addr   address );

CREATE INDEX  i_dept_addr1
          ON  department (dept_addr.city);
```

This code creates an index on the city attribute of the department address.

Wherever Oracle expects a column name in an index definition, you can also specify a scalar attribute of an object column.

Triggers

You can define triggers on an object table just as you can on other tables. You cannot define a trigger on the storage table for a nested table column or attribute.

You cannot modify the values of collections (or LOBs) in the code that defines a trigger action. Otherwise there are no special restrictions on using user-defined types with triggers.

The following example defines a trigger on the person_extent table defined in an earlier section:

```
CREATE TABLE movement (
     ssno        NUMBER,
     old_office  location,
     new_office  location );

CREATE TRIGGER trig1
  BEFORE UPDATE
              OF  office
              ON  person_extent
   FOR EACH ROW
           WHEN  new.office.city = 'REDWOOD SHORES'
```

```
BEGIN
  IF :new.office.building_no = 600 THEN
   INSERT INTO movement (ssno, old_office, new_office)
    VALUES (:old.ssno, :old.office, :new.office);
  END IF;
END;
```

Privileges on User-Defined Types and Their Methods

Privileges for user-defined types exist at the system level and schema object level.

System Privileges

Oracle defines the following system privileges for user-defined types:

- CREATE TYPE allows you to create user-defined types in your own schema.
- CREATE ANY TYPE allows you to create user-defined types in any schema.
- ALTER ANY TYPE allows you to alter user-defined types in any schema.
- DROP ANY TYPE allows you to drop named types in any schema.
- EXECUTE ANY TYPE allows you to use and reference named types in any schema.

The CONNECT and RESOURCE roles include the CREATE TYPE system privilege. The DBA role includes all of the above privileges.

Schema Object Privileges

The only schema object privilege that applies to user-defined types is EXECUTE.

EXECUTE on a user-defined type allows you to use the type to:

- Define a table.
- Define a column in a relational table.
- Declare a variable or parameter of the named type.

EXECUTE lets you invoke the type's methods, including the constructor.

Method execution and the associated permissions are the same as for stored PL/SQL procedures.

Using Types in New Types or Tables

In addition to the permissions detailed in the previous sections, you need specific privileges to:

- Create types or tables that use types created by other users.
- Grant use of your new types or tables to other users.

You must have the EXECUTE ANY TYPE system privilege, or you must have the EXECUTE object privilege for any type you use in defining a new type or table. You must have received these privileges explicitly, not through roles.

If you intend to grant access to your new type or table to other users, you must have either the required EXECUTE object privileges with the GRANT option or the EXECUTE ANY TYPE system privilege with the option WITH ADMIN OPTION. You must have received these privileges explicitly, not through roles.

Example

Assume that three users exist with the CONNECT and RESOURCE roles: user1, user2, and user3

User1 performs the following DDL in the user1 schema:

```
CREATE TYPE type1 AS OBJECT ( attr1 NUMBER );
CREATE TYPE type2 AS OBJECT ( attr2 NUMBER );
GRANT EXECUTE ON type1 TO user2;
GRANT EXECUTE ON type2 TO user2 WITH GRANT OPTION;
```

User2 performs the following DDL in the user2 schema:

```
CREATE TABLE tab1 OF user1.type1;
CREATE TYPE type3 AS OBJECT ( attr3 user1.type2 );
CREATE TABLE tab2 (col1 user1.type2 );
```

The following statements succeed, because user2 has EXECUTE on user1's type2 with the GRANT option:

```
GRANT EXECUTE ON type3 TO user3;
GRANT SELECT on tab2 TO user3;
```

However, the following grant fails, because user2 does not have EXECUTE on user1.type1 with the GRANT option:

```
GRANT SELECT ON tab1 TO user3;
```

User3 can successfully perform the following actions:

```
CREATE TYPE type4 AS OBJECT (attr4 user2.type3);
CREATE TABLE tab3 OF type4;
```

Privileges on Type Access and Object Access

The privileges that regulate use of tables apply equally to object tables:

- SELECT lets you access an object and its attributes from the table.
- UPDATE lets you modify attributes of objects in the table.
- INSERT lets you add new objects to the table.
- DELETE lets you delete objects from the table.

Similar table and column privileges regulate the use of table columns of user-defined types.

Retrieving data of user-defined types does not require type information. Interpreting the data, however, does require such information. When Oracle receives requests for type information, it verifies that the requestor has EXECUTE privilege on the type before supplying the requested information.

Consider the following schema:

```
CREATE TYPE emp_type (
  eno    NUMBER,
  ename  CHAR(31),
  eaddr  addr_t );

CREATE TABLE emp OF emp_type;
```

and the following two queries:

```
SELECT VALUE(e) FROM emp e;
SELECT eno, ename FROM emp;
```

For either query, Oracle checks the user's SELECT privilege for the emp table. For the first query, the user needs to obtain the emp_type type information to interpret the data. When the query accesses the emp_type type, Oracle checks the user's EXECUTE privilege.

Execution of the second query, however, does not involve named types, so Oracle does not check type privileges.

Additionally, using the schema from the previous section, user3 can perform the following queries:

```
SELECT tab1.col1.attr2 from user2.tab1 tab1;
```

```
SELECT t.attr4.attr3.attr2 FROM tab3 t;
```

Note that in both selects by user3, user3 does not have explicit privileges on the underlying types, but the statement succeeds because the type and table owners have the necessary privileges with the GRANT option.

Oracle checks privileges on the following requests, and returns an error if the requestor does not have the privilege for the action:

- Pinning an object in the object cache using its REF value causes Oracle to check SELECT privilege on the containing object table.
- Modifying an existing object or flushing an object from the object cache, causes Oracle to check UPDATE privilege on the destination object table. Flushing a new object causes Oracle to check INSERT privilege on the destination object table.
- Deleting an object causes Oracle to check DELETE privilege on the destination table. Pinning an object of named type causes Oracle to check EXECUTE privilege on the object type.
- Invoking a method causes Oracle to check EXECUTE privilege on the corresponding object type.

Oracle does not provide column level privileges for object tables.

Dependencies and Incomplete Types

Types can depend upon each other for their definitions. For example, you might want to define object types `employee` and `department` in such a way that one attribute of `employee` is the department the employee belongs to and one attribute of `department` is the employee who manages the department.

Types that depend on each other in this way, either directly or via intermediate types, are called *mutually dependent*. A diagram of mutually dependent types, with arrows representing the dependencies, always reveals a path of arrows starting and ending at one of the types.

Oracle allows such cyclic dependencies only when at least one branch of the cycle uses REFs.

For example, you can define the following types:

```
CREATE TYPE department;

CREATE TYPE employee AS OBJECT (
  name    VARCHAR2(30),
```

```
  dept    REF department,
  supv    REF employee );

CREATE TYPE emp_list AS TABLE OF employee;

CREATE TYPE department AS OBJECT (
  name    VARCHAR2(30),
  mgr     REF employee,
  staff   emp_list );
```

This is a legal set of mutually dependent types and a legal sequence of SQL DDL statements. Oracle compiles it without errors. The first statement

```
CREATE TYPE department;
```

is optional. It makes the compilation proceed without errors. It establishes `department` as an *incomplete object type*. A REF to an incomplete object type compiles without error, so the compilation of `employee` proceeds.

When Oracle reaches the last statement, which completes the definition of `department`, all of the components of `department` have compiled successfully, so the compilation finishes without errors.

Without the optional declaration of `department` as an incomplete type, `employee` compiles with errors. Oracle then automatically adds `employee` to its library of schema objects as an incomplete object type. This makes the declarations of `emp_list` and `department` compile without errors. When `employee` is recompiled after `emp_list` and `department` are complete, `employee` compiles without errors and becomes a complete object type.

Completing Incomplete Types

Once you have declared an incomplete object type, you must complete it as an object type. You cannot, for example, declare it to be a table type or an array type. The only alternative is to drop the type.

This is also true if Oracle has made the type an incomplete object type for you — as it did when `employee` failed to compile in the previous section.

This restriction applies even if there are no REFs to the incomplete object type anywhere in the schema. The rule Oracle follows is that once it flags a type as the potential target of REFs, that type must remain a potential REF target until it is dropped.

Oracle recognizes only object types as potential REF targets.

Type Dependencies of Tables

If a table contains data that relies on a type definition for access, any change to the type causes the table's data to become inaccessible. This happens if privileges required by the type are revoked or if the type or a type it depends on is dropped. The table then becomes invalid and cannot be accessed.

A table that is invalid because of missing privileges automatically becomes valid and accessible if the required privileges are re-granted.

A table that is invalid because a type it depends on has been dropped can never be accessed again. The only permissible action is to drop the table.

The SQL commands REVOKE and DROP TYPE return an error and abort if the type referred to in the command has tables or other types that depend on it.

The FORCE option with either of these commands overrides that behavior. The command succeeds and the affected tables or types become invalid.

Import/Export of User-Defined Types

The Export and Import utilities move data into and out of Oracle databases. They are also back up or archive data and aid migration to different releases of the Oracle RDBMS.

Export and Import support user-defined types. Export writes user-defined type definitions, foreign function library definitions, directory alias definitions, and all of the associated data to the dump file. Import then recreates these items from the dump file.

> **Additional Information:** See *Oracle8 Utilities* for more information about Export and Import.

13

Object Views

The choice of a point of view is the initial act of a culture.

José Ortega y Gasset, *The Modern Theme*

This chapter describes object views. It contains the following major sections:

- Introduction
- Defining Object Views
- Using Object Views
- Updating Object Views

> **Attention:** The features described in this chapter are available only if you have purchased Oracle8 Enterprise Edition with the Objects Option. Wherever the term Oracle server appears in this chapter it refers to Oracle8 Enterprise Edition with the Objects Option. See *Getting to Know Oracle8 and the Oracle8 Enterprise Edition* for information about the features and options available with Oracle8 Enterprise Edition.

Introduction

Just as a view is a virtual table, an *object view* is a virtual object table.

Oracle provides object views as an extension of the basic relational view mechanism. By using object views, you can create virtual object tables from data — of either built-in or user-defined types — stored in the columns of relational or object tables in the database.

Object views provide the ability to offer specialized or restricted access to the data and objects in a database. For example, you might use an object view to provide a version of an employee object table that doesn't have attributes containing sensitive data and doesn't have a deletion method.

Object views allow the use of relational data in object-oriented applications. They let users

- Try object-oriented programming techniques without converting existing tables.
- Convert data gradually and transparently from relational tables to object-relational tables.
- Use legacy RDBMS data with existing object-oriented applications.

Advantages of Object Views

Using object views can lead to better performance. Relational data that make up a row of an object view traverse the network as a unit, potentially saving many round trips.

You can fetch relational data into the client-side object cache and map it into *C or C++* structures so 3GL applications can manipulate it just like native structures.

Object views provide a gradual migration path for legacy data.

Object views provide for co-existence of relational and object-oriented applications. They make it easier to introduce object-oriented applications to existing relational data without having to make a drastic change from one paradigm to another.

Object views provide the flexibility of looking at the same relational or object data in more than one way. Thus you can use different in-memory object representations for different applications without changing the way you store the data in the database.

Defining Object Views

Conceptually, the procedure for defining an object view is simple:

1. Define an object type to be represented by rows of the object view.
2. Write a query that specifies which data in which relational tables contain the attributes for objects of that type.
3. Specify an object identifier, based on attributes of the underlying data, to allow REFs to the objects (rows) of the object view.

The object identifier corresponds to the unique object identifier that Oracle generates automatically for rows of object tables. In the case of object views, however, the declaration must specify something that is unique in the underlying data (for example, a primary key).

If the object view is based on a table or another object view and you don't specify an object identifier, Oracle uses the object identifier from the original table or object view.

If you wish to be able to update a complex object view, you may have to take another step:

4. Write an INSTEAD OF trigger procedure (see "Updating Object Views" on page 13-4) for Oracle to execute whenever an application program tries to update data in the object view.

After these steps you can use an object view just like an object table.

For example, the following SQL statements define an object view:

```
CREATE TABLE emp_table  (
    empnum   NUMBER (5),
    ename    VARCHAR2 (20),
    salary   NUMBER (9, 2),
    job      VARCHAR2 (20) );

CREATE TYPE employee_t  (
    empno    NUMBER (5),
    ename    VARCHAR2 (20),
    salary   NUMBER (9, 2),
    job      VARCHAR2 (20) );

CREATE VIEW emp_view1 OF employee_t
    WITH OBJECT OID (empno) AS
        SELECT   e.empnum, e.ename, e.salary, e.job
        FROM     emp_table e
        WHERE    job = 'Developer';
```

The object view looks to the user like an object table whose underlying type is `employee_t`. Each row contains an object of type `employee_t`. Each row has a unique object identifier.

Oracle constructs the object identifier based on the specified key. In most cases it is the primary key of the base table. If the query that defines the object view involves joins, however, you must provide a key across all tables involved in the joins, so that the key still uniquely identifies rows of the object view.

Note: Columns in the WITH OBJECT OID clause — `empno` in the example — must also be attributes of the underlying object type — `employee_t` in the example. This makes it easy for trigger programs to identify the corresponding row in the base table uniquely.

Using Object Views

Data in the rows of an object view may come from more than one table, but the object still traverses the network in one operation. When the instance is in the client side object cache, it appears to the programmer as a C or C++ structure or as a PL/SQL object variable. You can manipulate it like any other native structure.

You can refer to object views in SQL statements the same way you refer to an object table. For example, object views can appear in a SELECT list, in an UPDATE-SET clause, or in a WHERE clause.

You can also define object views on object views.

You can access object view data on the client side using the same OCI calls you use for objects from object tables. For example, you can use *OCIObjectPin()* for pinning a REF and *OCIObjectFlush()* for flushing an object to the server. When you update or flush to the server an object in an object view, Oracle updates the object view.

> **Additional Information:** See *Oracle Call Interface Programmer's Guide* for more information about OCI calls.

Updating Object Views

You can update, insert, and delete the data in an object view using the same SQL DML you use for object tables. Oracle updates the base tables of the object view if there is no ambiguity.

A view is not updatable if its view query contains joins, set operators, group functions, GROUP BY, or DISTINCT. If a view query contains pseudocolumns or expres-

sions, the corresponding view columns are not updatable. Object views often involve joins.

To overcome these obstacles Oracle provides *INSTEAD OF triggers* (see Chapter 18, "Database Triggers"). They are called INSTEAD OF triggers because Oracle executes the trigger body instead of the actual DML statement.

INSTEAD OF triggers provide a transparent way to update object views or relational views. You write the same SQL DML (INSERT, DELETE, and UPDATE) statements as for an object table. Oracle invokes the appropriate trigger instead of the SQL statement, and the actions specified in the trigger body take place.

> **Additional Information:** See *Oracle8 Application Developer's Guide* for a purchase order/line item example that uses an INSTEAD OF trigger.

Part V

Data Access

Part V describes how to use transactions consisting of SQL statements to access data in an Oracle database, and it describes procedural language constructs that provide additional functionality for data access. It also describes the optimizer, which chooses the most efficient way to execute each SQL statement.

Part V contains the following chapters:

- Chapter 14, “SQL and PL/SQL”
- Chapter 15, “Transaction Management”
- Chapter 16, “Advanced Queuing”
- Chapter 17, “Procedures and Packages”
- Chapter 18, “Database Triggers”
- Chapter 19, “Oracle Dependency Management”
- Chapter 20, “The Optimizer”

14

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)
- SQL Processing
- PL/SQL

> **Additional Information:** 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 nonprocedural language; users describe in SQL what they want done, and the SQL language compiler automatically generates a procedure to navigate the database and perform the desired task.

IBM Research developed and defined SQL, and ANSI/ISO has refined SQL as the standard language for relational database management systems. The 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, Oracle Enterprise Manager, 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.

> **Additional Information:** See the *Oracle8 SQL Reference* for detailed information about SQL commands and other parts of SQL (such as operators, functions, and format models).
>
> See the *Oracle Enterprise Manager Administrator's Guide* for information about Oracle Enterprise Manager and Server Manager commands, including their distinction from SQL commands.

This section includes the following topics:

- SQL Statements
- Identifying Nonstandard 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 consists partially of SQL *reserved words*, which have special meaning in SQL and cannot be used for any other purpose. For example, SELECT and UPDATE are reserved words and cannot be used as table names.

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

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

Note: Oracle also supports the use of SQL statements in PL/SQL program units; see Chapter 17, "Procedures and Packages" and Chapter 18, "Database Triggers" for more information about this feature.

Data Manipulation Language (DML) Statements

DML statements query or manipulate data in existing schema objects. They enable you to

- retrieve data from one or more tables or views (SELECT)
- add new rows of data into a table or view (INSERT)

- change column values in existing rows of a table or view (UPDATE)
- remove rows from tables or views (DELETE)
- see the execution plan for a SQL statement (EXPLAIN PLAN)
- lock a table or view, temporarily limiting other users' access (LOCK TABLE)

DML statements are the most frequently used SQL statements. Some examples of DML statements 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');
```

Data Definition Language (DDL) Statements

DDL statements define, alter the structure of, and drop schema objects. DDL statements enable you to

- create, alter, and drop schema objects and other database structures, including the database itself and database users (CREATE, ALTER, DROP)
- change the names of schema objects (RENAME)
- delete all the data in schema objects without removing the objects' structure (TRUNCATE)
- gather statistics about schema objects, validate object structure, and list chained rows within objects (ANALYZE)
- grant and revoke privileges and roles (GRANT, REVOKE)
- turn auditing options on and off (AUDIT, NOAUDIT)
- add a comment to the data dictionary (COMMENT)

DDL statements implicitly commit the preceding and start a new transaction.

Some examples of DDL statements 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 25, “Controlling Database Access”, Chapter 26, “Privileges and Roles”, and Chapter 27, “Auditing”.

Transaction Control Statements

Transaction control statements manage the changes made by DML statements and group DML statements into transactions. They enable you to

- make a transaction’s changes permanent (COMMIT)
- undo the changes in a transaction, either since the transaction started or since a savepoint (ROLLBACK)
- set a point to which you can roll back (SAVEPOINT)
- establish properties for a transaction (SET TRANSACTION)

Session Control Statements

Session control statements manage the properties of a particular user’s session. For example, they enable you to

- alter the current session by performing a specialized function, such as enabling and disabling the SQL trace facility (ALTER SESSION)
- enable and disable roles (groups of privileges) for the current session (SET ROLE)

System Control Statements

System control statements change the properties of the Oracle server instance.

The only system control command is ALTER SYSTEM. It enables you to change settings (such as the minimum number of shared servers), to kill a session, and to perform other tasks.

Embedded SQL Statements

Embedded SQL statements incorporate DDL, DML, and transaction control statements within a procedural language program. They are used with the Oracle precompilers.

Embedded SQL statements enable you to

- define, allocate, and release cursors (DECLARE CURSOR, OPEN, CLOSE)
- specify a database and connect to Oracle (DECLARE DATABASE, CONNECT)
- assign variable names (DECLARE STATEMENT)
- initialize descriptors (DESCRIBE)
- specify how error and warning conditions are handled (WHENEVER)
- parse and execute SQL statements (PREPARE, EXECUTE, EXECUTE IMMEDIATE)
- retrieve data from the database (FETCH).

Identifying Nonstandard SQL

Oracle provides extensions to the standard SQL "Database Language with Integrity Enhancement". The Federal Information Processing Standard for SQL (FIPS 127-2) requires vendors to supply a method for identifying SQL statements that use such extensions. You can identify or "flag" Oracle extensions in interactive SQL, the Oracle precompilers, or SQL*Module by using the FIPS flagger.

If you are concerned with the portability of your applications to other implementations of SQL, use the FIPS flagger.

> **Additional Information:** For information on how to use the FIPS flagger, see the *Pro*C/C++ Precompiler Programmer's Guide*, *Pro*COBOL Precompiler Programmer's Guide*, or *SQL*Module for Ada Programmer's Guide*.

Recursive SQL

When a DDL statement is issued, Oracle implicitly issues *recursive SQL statements* that modify data dictionary information. Users need not be concerned with the recursive SQL internally performed by Oracle.

Cursors

A cursor is a handle or name for a *private SQL area* — an area in memory in which a parsed statement and other information for processing the statement are kept.

Although most Oracle users rely on the automatic cursor handling of the Oracle utilities, the programmatic interfaces offer application designers more control over cursors. In application development, a cursor is a named resource available to a

program and can be used specifically for the parsing of SQL statements embedded within the application.

Each user session can open multiple cursors up to the limit set by the initialization parameter OPEN_CURSORS. However, applications should close unneeded cursors to conserve system memory. If a cursor cannot be opened due to a limit on the number of cursors, the database administrator can alter the OPEN_CURSORS initialization parameter.

Some statements (primarily DDL statements) require Oracle to implicitly issue recursive SQL statements, which also require *recursive cursors*. For example, a CREATE TABLE statement causes many updates to various data dictionary tables to record the new table and columns. *Recursive calls* are made for those recursive cursors; one cursor may execute several recursive calls. These recursive cursors also use *shared SQL areas*.

Shared SQL

Oracle automatically notices when applications send identical SQL statements to the database. The SQL area used to process the first occurrence of the statement is *shared* — that is, used for processing subsequent occurrences of that same statement. Therefore, only one shared SQL area exists for a unique statement. Since shared SQL areas are shared memory areas, any Oracle process can use a shared SQL area. The sharing of SQL areas reduces memory usage on the database server, thereby increasing system throughput.

In evaluating whether statements are identical, Oracle considers SQL statements issued directly by users and applications as well as recursive SQL statements issued internally by a DDL statement.

> **Additional Information:** See the *Oracle8 Application Developer's Guide* for more information on shared SQL.

Parsing

Parsing is one stage in the processing of a SQL statement. When an application issues a SQL statement, the application makes a parse call to Oracle. During the parse call, Oracle

- checks the statement for syntactic and semantic validity
- determines whether the process issuing the statement has privileges to execute it
- allocates a private SQL area for the statement

Oracle also determines whether there is an existing shared SQL area containing the parsed representation of the statement in the library cache. If so, the user process uses this parsed representation and executes the statement immediately. If not, Oracle generates the parsed representation of the statement, and the user process allocates a shared SQL area for the statement in the library cache and stores its parsed representation there.

Note the difference between an *application* making a parse call for a SQL statement and Oracle actually parsing the statement. A *parse call* by the *application* associates a SQL statement with a private SQL area. 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.

Both parse calls and parsing can be expensive relative to execution, so it is desirable to perform them as seldom as possible.

This discussion applies also to the parsing of PL/SQL blocks and allocation of PL/SQL areas. (See "PL/SQL" on page 14-15.) 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.

SQL Processing

This section introduces the basics of SQL processing. Topics include:

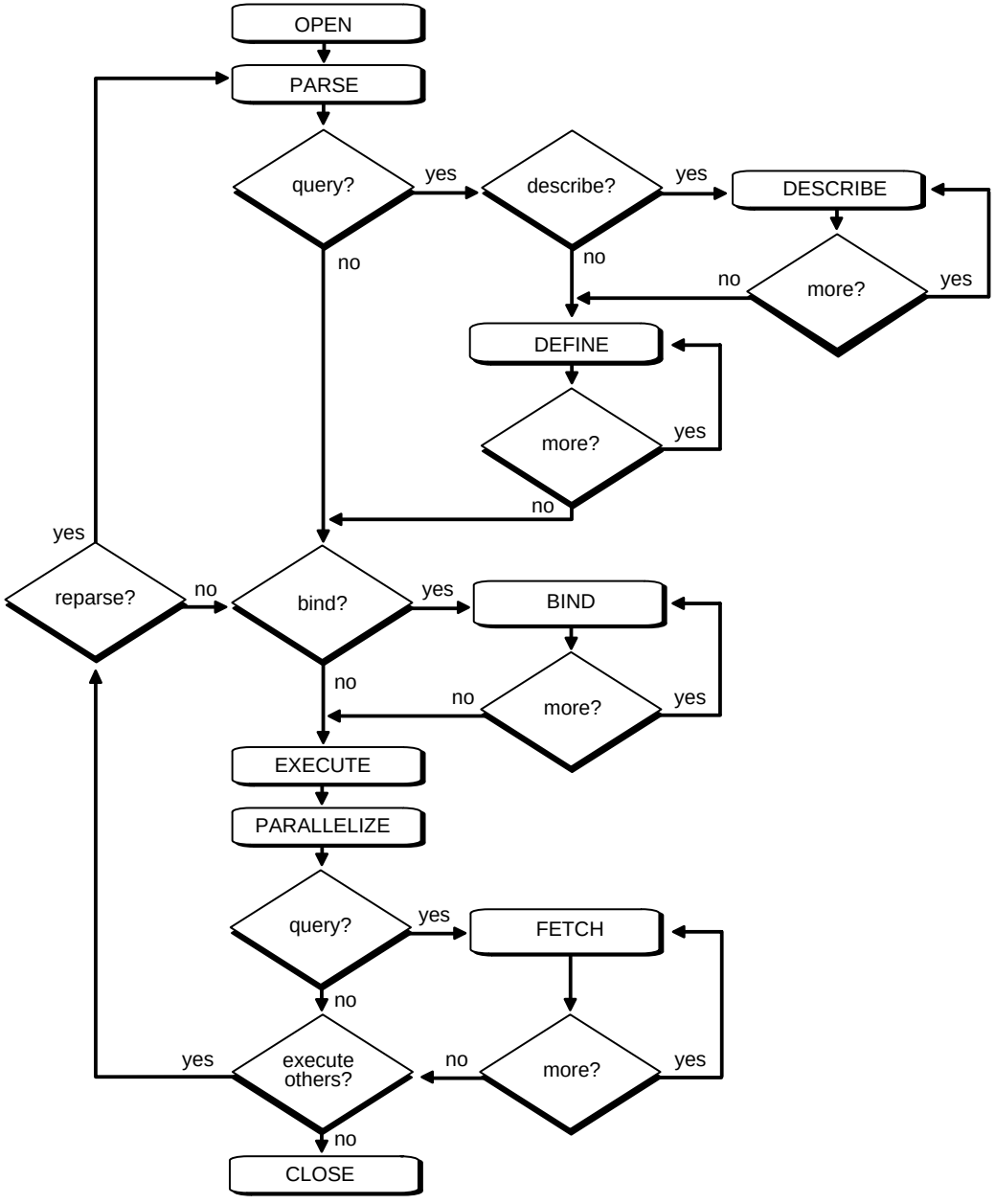
- Overview of SQL Statement Execution
- DML Statement Processing
- DDL Statement Processing
- Controlling Transactions

Overview of SQL Statement Execution

Figure 14–1 outlines the stages commonly used to process and execute a SQL statement. In some cases, Oracle might execute these stages in a slightly different order. For example, the DEFINE stage could occur just before the FETCH stage, depending on how you wrote your code.

For many Oracle tools, several of the stages are performed automatically. Most users need not be concerned with or aware of this level of detail. However, you might find this information useful when writing Oracle applications.

Figure 14–1 The Stages in Processing a SQL Statement



DML Statement Processing

This section provides a simple example of what happens during the execution of a SQL statement, in each stage of DML statement processing.

Assume that you are using a Pro*C program to increase the salary for all employees in a department. Also assume that the program you are using has connected to Oracle and that you are connected to the proper schema to update the EMP table. You might embed the following SQL statement in your program:

```
EXEC SQL UPDATE emp SET sal = 1.10 * sal
    WHERE deptno = :dept_number;
```

DEPT_NUMBER is a program variable containing a value for department number. When the SQL statement is executed, the value of DEPT_NUMBER is used, as provided by the application program.

The following stages are necessary for each type of statement processing:

- Stage 1: Create a Cursor
- Stage 2: Parse the Statement
- Stage 5: Bind Any Variables
- Stage 7: Execute the Statement
- Stage 9: Close the Cursor

Optionally, you can include another stage:

- Stage 6: Parallelize the Statement

Queries (SELECTs) require several additional stages, as shown in Figure 14–1:

- Stage 3: Describe Results of a Query
- Stage 4: Define Output of a Query
- Stage 8: Fetch Rows of a Query
- Stage 9: Close the Cursor

See “Query Processing” on page 14-11 for more information.

Stage 1: Create a Cursor

A program interface call creates a cursor. The cursor is created independent of any SQL statement; it is created in expectation of any SQL statement. In most applications, cursor creation is automatic. However, in precompiler programs, cursor creation can either occur implicitly or be explicitly declared.

Stage 2: Parse the Statement

During parsing, the SQL statement is passed from the user process to Oracle, and a parsed representation of the SQL statement is loaded into a shared SQL area. Many errors can be caught during this stage of statement processing.

Parsing is the process of:

- translating a SQL statement, verifying it to be a valid statement
- performing data dictionary lookups to check table and column definitions
- acquiring parse locks on required objects so that their definitions do not change during the statement's parsing
- checking privileges to access referenced schema objects
- determining the optimal execution plan for the statement
- loading it into a shared SQL area
- for distributed statements, routing all or part of the statement to remote nodes that contain referenced data

Oracle parses a SQL statement only if a shared SQL area for an identical SQL statement does not exist in the shared pool. In this case, a new shared SQL area is allocated and the statement is parsed. (See "Shared SQL" on page 14-7.)

The parse stage includes processing requirements that need to be done only once no matter how many times the statement is executed. Oracle translates each SQL statement only once, reexecuting that parsed statement during subsequent references to the statement.

Although the parsing of a SQL statement validates that statement, parsing only identifies errors that can be found **before statement execution.** Thus, some errors cannot be caught by parsing. For example, errors in data conversion or errors in data (such as an attempt to enter duplicate values in a primary key) and deadlocks are all errors or situations that can be encountered and reported only during the execution stage.

Query Processing

Queries are different from other types of SQL statements because, if successful, they return data as results. Whereas other statements simply return success or failure, a query can return one row or thousands of rows. The results of a query are **always in tabular format,** and the rows of the result are *fetched* (retrieved), either a row at a time or in groups.

Several issues relate only to query processing. Queries include not only explicit SELECT statements but also the implicit queries *(subqueries)* in other SQL statements. For example, each of the following statements requires a query as a part of its execution:

```
INSERT INTO table SELECT...

UPDATE table SET x = y WHERE...

DELETE FROM table WHERE...

CREATE table AS SELECT...
```

In particular, queries:

- require *read consistency*
- can use temporary segments for intermediate processing
- can require the describe, define, and fetch stages of SQL statement processing.

Stage 3: Describe Results of a Query

The describe stage is necessary only if the characteristics of a query's result are not known; for example, when a query is entered interactively by a user.

In this case, the describe stage determines the characteristics (datatypes, lengths, and names) of a query's result.

Stage 4: Define Output of a Query

In the define stage for queries, you specify the location, size, and datatype of variables defined to receive each fetched value. Oracle performs datatype conversion if necessary.

Stage 5: Bind Any Variables

At this point, Oracle knows the meaning of the SQL statement but still does not have enough information to execute the statement. Oracle needs values for any variables listed in the statement; in the example, Oracle needs a value for DEPT_NUMBER. The process of obtaining these values is called *binding variables.*

A program must specify the location (memory address) where the value can be found. End users of applications might be unaware that they are specifying bind variables, because the Oracle utility might simply prompt them for a new value.

Because you specify the location (binding by reference), you need not rebind the variable before re-execution. You can change its value and Oracle looks up the value on each execution, using the memory address.

You must also specify a datatype and length for each value (unless they are implied or defaulted) if Oracle needs to perform datatype conversion.

> **Additional Information:** For more information about specifying a datatype and length for a value, refer to the following publications:
>
> - *Oracle Call Interface Programmer's Guide*
> - *Pro*C/C++ Precompiler Programmer's Guide* (see "Dynamic SQL Method 4")
> - *Pro*COBOL Precompiler Programmer's Guide* (see "Dynamic SQL Method 4")

Stage 6: Parallelize the Statement

Oracle can parallelize queries (SELECTs), INSERTs, UPDATEs, DELETEs, and some DDL operations such as index creation, creating a table with a subquery, and operations on partitions. Parallelization causes multiple server processes to perform the work of the SQL statement so that it can complete faster.

See Chapter 22, "Parallel Execution", for more information about parallel SQL.

Stage 7: Execute the Statement

At this point, Oracle has all necessary information and resources, so the statement is executed. If the statement is a query or an INSERT statement, no rows need to be locked because no data is being changed. If the statement is an UPDATE or DELETE statement, however, all rows that the statement affects are locked from use by other users of the database until the next COMMIT, ROLLBACK, or SAVEPOINT for the transaction. This ensures data integrity.

For some statements you can specify a number of executions to be performed. This is called *array processing*. Given *n* number of executions, the bind and define locations are assumed to be the beginning of an array of size *n*.

Stage 8: Fetch Rows of a Query

In the fetch stage, rows are selected and ordered (if requested by the query), and each successive fetch retrieves another row of the result, until the last row has been fetched.

Stage 9: Close the Cursor

The final stage of processing a SQL statement is closing the cursor.

DDL Statement Processing

The execution of DDL statements differs from the execution of DML statements and queries because the success of a DDL statement requires write access to the data dictionary. For these statements, parsing (Stage 2) actually includes parsing, data dictionary lookup, and execution.

Transaction management, session management, and system management SQL statements are processed using the parse and execute stages. To reexecute them, simply perform another execute.

Controlling Transactions

In general, only application designers using the programming interfaces to Oracle are concerned with the types of actions that should be grouped together as one transaction. Transactions must be defined properly so that work is accomplished in logical units and data is kept consistent. A transaction should consist of all of the necessary parts for one logical unit of work, no more and no less.

- Data in all referenced tables should be in a consistent state before the transaction begins and after it ends.
- Transactions should consist of only the SQL statements that make one consistent change to the data.

For example, a transfer of funds between two accounts (the transaction or logical unit of work) should include the debit to one account (one SQL statement) and the credit to another account (one SQL statement). Both actions should either fail or succeed together as a unit of work; the credit should not be committed without the debit. Other nonrelated actions, such as a new deposit to one account, should not be included in the transfer of funds transaction.

In addition to determining which types of actions form a transaction, when you design an application you must also determine when it is useful to use the BEGIN_DISCRETE_TRANSACTION procedure to improve the performance of short, non-distributed transactions. See “Discrete Transaction Management” on page 15-8 for more information.

PL/SQL

PL/SQL is Oracle's procedural language extension to SQL. PL/SQL enables you to mix SQL statements with procedural constructs. With PL/SQL, you can define and execute PL/SQL program units such as procedures, functions, and packages.

PL/SQL program units generally are categorized as anonymous blocks and stored procedures.

An *anonymous block* is a PL/SQL block that appears within your application and it is not named or stored in the database. In many applications, PL/SQL blocks can appear wherever SQL statements can appear.

A *stored procedure* is a PL/SQL block that Oracle stores in the database and can be called by name from an application. When you create a stored procedure, Oracle parses the procedure and stores its parsed representation in the database. Oracle also allows you to create and store functions (which are similar to procedures) and packages (which are groups of procedures and functions).

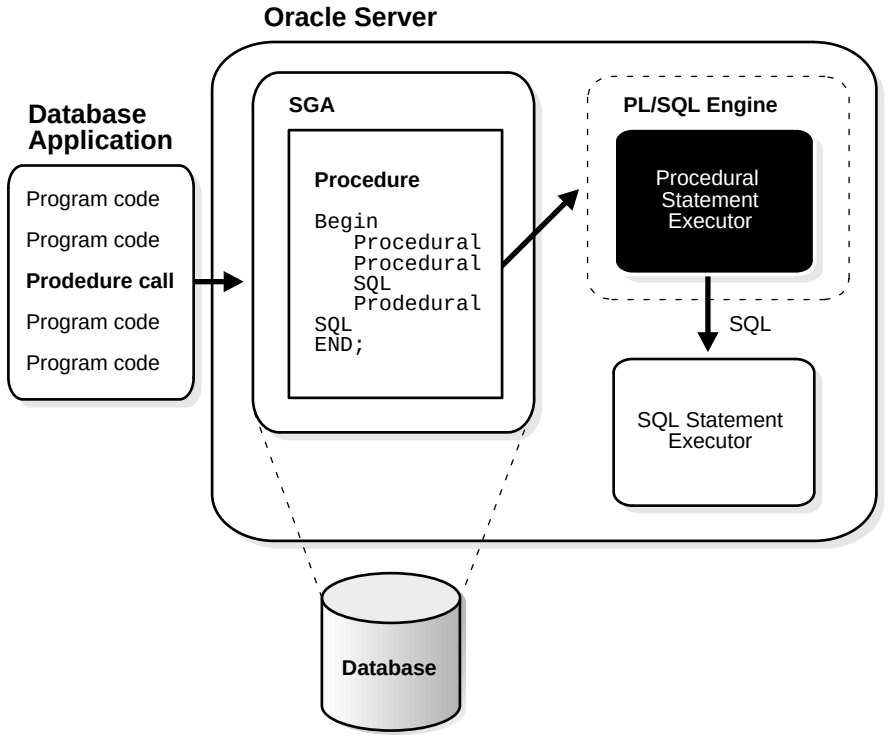
For information on stored procedures, functions, packages, and database triggers, see Chapter 17, "Procedures and Packages", and Chapter 18, "Database Triggers".

How PL/SQL Executes

The *PL/SQL engine*, which processes PL/SQL program units, is a special component of many Oracle products, including the Oracle server.

Figure 14–2 illustrates the PL/SQL engine contained in Oracle server.

Figure 14–2 The PL/SQL Engine and the Oracle Server



The procedure (or package) is stored in a database. When an application calls a procedure stored in the database, Oracle loads the compiled procedure (or package) into the shared pool in the system global area (SGA), and the PL/SQL and SQL statement executors work together to process the statements within the procedure.

The following Oracle products contain a PL/SQL engine:

- Oracle server
- Oracle Forms (Version 3 and later)
- SQL*Menu (Version 5 and later)
- Oracle Reports (Version 2 and later)
- Oracle Graphics (Version 2 and later)

You can call a stored procedure from another PL/SQL block, which can be either an anonymous block or another stored procedure. For example, you can call a stored procedure from Oracle Forms (Version 3 or later).

Also, you can pass anonymous blocks to Oracle from applications developed with these tools:

- Oracle precompilers (including user exits)
- Oracle Call Interfaces (OCIs)
- SQL*Plus
- Server Manager
- Oracle Enterprise Manager

Language Constructs for PL/SQL

PL/SQL blocks can include the following PL/SQL language constructs:

- variables and constants
- cursors
- exceptions

This section gives a general description of each construct.

Additional Information: See the *PL/SQL User's Guide and Reference*.

Variables and Constants

Variables and constants can be declared within a procedure, function, or package. A variable or constant can be used in a SQL or PL/SQL statement to capture or provide a value when one is needed.

Note: Some interactive tools, such as Server Manager, allow you to define variables in your current session. You can use such variables just as you would variables declared within procedures or packages.

Cursors

Cursors can be declared explicitly within a procedure, function, or package to facilitate record-oriented processing of Oracle data. Cursors also can be declared implicitly (to support other data manipulation actions) by the PL/SQL engine.

Exceptions

PL/SQL allows you to explicitly handle internal and user-defined error conditions, called *exceptions*, that arise during processing of PL/SQL code. Internal exceptions are caused by illegal operations, such as division by zero, or Oracle errors returned to the PL/SQL code. User-defined exceptions are explicitly defined and signaled within the PL/SQL block to control processing of errors specific to the application (for example, debiting an account and leaving a negative balance).

When an exception is *raised* (signaled), the normal execution of the PL/SQL code stops, and a routine called an exception handler is invoked. Specific exception handlers can be written to handle any internal or user-defined exception.

Stored Procedures

Oracle also allows you to create and call stored procedures. If your application calls a stored procedure, the parsed representation of the procedure is retrieved from the database and processed by the PL/SQL engine in Oracle.

Note: While many Oracle products have PL/SQL components, this manual specifically covers only the procedures and packages that can be stored in an Oracle database and processed using the PL/SQL engine of the Oracle server.

Additional Information: The PL/SQL capabilities of each Oracle tool are described in the appropriate tool user guide.

You can call stored procedures from applications developed using these tools:

- Oracle precompilers (including user exits)
- Oracle Call Interfaces (OCIs)
- SQL*Module
- SQL*Plus
- Server Manager
- Oracle Enterprise Manager

You can also call a stored procedure from another PL/SQL block, either an anonymous block or another stored procedure. See Chapter 17, “Procedures and Packages” for more information.

Additional Information: For information on how to call stored procedures from each type of application, see the documentation for the specific application tool, such as the *Pro*C/C++ Precompiler Programmer's Guide* or *Pro*COBOL Precompiler Programmer's Guide*.

Dynamic SQL in PL/SQL

You can write stored procedures and anonymous PL/SQL blocks that include dynamic SQL by using the DBMS_SQL package. Dynamic SQL statements are not embedded in your source program; rather, they are stored in character strings that are entered into, or built by, the program at runtime.

This enables 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 DDL 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.

Additional Information: For more information about dynamic SQL, see the *Oracle8 Application Developer's Guide*.

External Procedures

A PL/SQL procedure executing on an Oracle server can call an external procedure or function that is written in the C programming language and stored in a shared library. The C routine executes in a separate address space from that of the Oracle server.

Additional Information: See the *PL/SQL User's Guide and Reference* for detailed information about external procedures.

15

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).

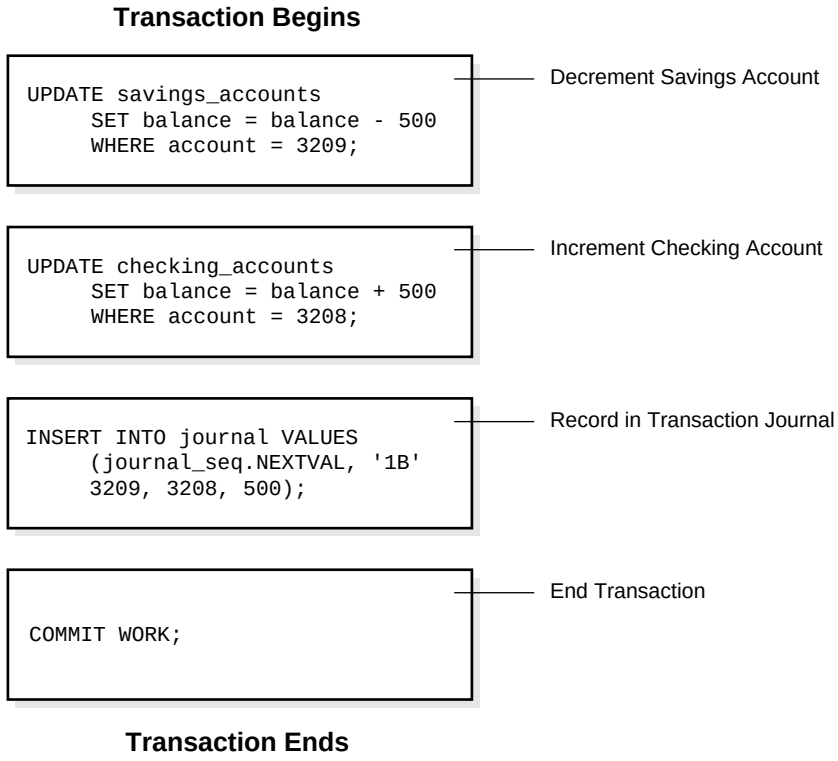
A transaction begins with the first executable SQL statement. A transaction ends when it is committed or rolled back, either explicitly (with a COMMIT or ROLLBACK statement) or implicitly (when a DDL statement is issued).

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 15–1 illustrates the banking transaction example.

Figure 15–1 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 multirow update are changed). However, until the transaction that contains the statement is committed, the transaction can be rolled back, and all of the changes of the statement can be undone. A statement, rather than a transaction, executes successfully.

Committing means that a user has said either explicitly or implicitly "make the changes in this transaction permanent". The changes made by the SQL statement(s)

of your transaction become permanent and visible to other users only after your transaction has been committed. Only other users' transactions that started after yours will see the committed changes.

Statement-Level Rollback

If at any time during execution a SQL statement causes an error, all effects of the statement are rolled back. The effect of the rollback is as if that statement were never executed. This is a *statement-level rollback*.

Errors discovered during SQL statement *execution* cause statement-level rollbacks. (An example of such an error is attempting to insert a duplicate value in a primary key.) Errors discovered during SQL statement *parsing* (such as a syntax error) have not yet been executed, so do not cause a statement-level rollback. Single SQL statements involved in a *deadlock* (competition for the same data) may also cause a statement-level rollback. See "Deadlocks" on page 23-16.

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.

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 2-18 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 modifies data is committed, the following has occurred:

- Oracle has generated rollback segment records in rollback segment buffers of the system global area (SGA). The rollback information contains the old data values changed by the SQL statements of the transaction.
- Oracle has generated redo log entries in the redo log buffer of the SGA. These changes may go to disk before a transaction is committed.
- The changes have been made to the database buffers of the SGA. These changes may go to disk before a transaction actually is committed.

> **Note:** The data changes for a committed transaction, stored in the database buffers of the SGA, are not necessarily written immediately to the datafiles by the database writer (DBW*n*) background process. This writing takes place when it is most efficient to do so. It may happen before the transaction commits or, alternatively, it may happen some time after the transaction commits.

When a transaction is committed, the following occurs:

- 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.
- The log writer process (LGWR) writes redo log entries in the SGA's redo log buffers to the online redo log file; it also writes the transaction's SCN to the online redo log file. This atomic event constitutes the commit of the transaction.

- Oracle releases locks held on rows and tables. (See “Locking Mechanisms” on page 23-3 for a discussion of locks.)
- Oracle marks the transaction “complete”.

See “Oracle Processes” on page 7-5 for more information about the background processes 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 “Savepoints” on page 15-7 for a complete explanation.

All types of rollbacks use the same procedures:

- statement-level rollback (due to statement or deadlock execution error)
- rollback to a savepoint
- rollback of a transaction due to user request
- rollback of a transaction due to abnormal process termination
- rollback of all outstanding transactions when an instance terminates abnormally
- rollback of incomplete transactions during recovery

In rolling back **an entire transaction**, without referencing any savepoints, the following occurs:

- 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 23-3 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 23-3 for a discussion of locks).
- The transaction remains active and can be continued.

Savepoints

You can declare intermediate markers called *savepoints* within the context of a transaction. Savepoints divide a long transaction into smaller parts.

Using savepoints, you can arbitrarily mark your work at any point within a long transaction. You then have the option later of rolling back work performed before the current point in the transaction (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 similarly 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 reexecute the function with revised parameters or perform a recovery action.

After a rollback to a savepoint, Oracle releases the data locks obtained by rolled back statements. Other transactions that were waiting for the previously locked resources can proceed. Other transactions that want to update previously locked rows can do so.

The Two-Phase Commit Mechanism

In a distributed database, Oracle must coordinate transaction control over a network and maintain data consistency, even if a network or system failure occurs.

A *two-phase commit* mechanism guarantees that *all* database servers participating in a distributed transaction either all commit or all roll back the statements in the transaction. A two-phase commit mechanism also protects implicit DML operations performed by integrity constraints, remote procedure calls, and triggers.

The Oracle two-phase commit mechanism is completely transparent to users who issue distributed transactions. In fact, users need not even know the transaction is distributed. A COMMIT statement denoting the end of a transaction automatically triggers the two-phase commit mechanism to commit the transaction; no coding or complex statement syntax is required to include distributed transactions within the body of a database application.

The recoverer (RECO) background process automatically resolves the outcome of *in-doubt distributed transactions* — distributed transactions in which the commit was interrupted by any type of system or network failure. After the failure is repaired and communication is reestablished, the RECO of each local Oracle server automatically commits or rolls back any in-doubt distributed transactions consistently on all involved nodes.

In the event of a long-term failure, Oracle allows each local administrator to manually commit or roll back any distributed transactions that are in doubt as a result of the failure. This option enables the local database administrator to free up any locked resources that may be held indefinitely as a result of the long-term failure.

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

Discrete Transaction Management

Application developers can improve the performance of short, nondistributed 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 discrete 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.

> **Additional Information:** For more information on discrete transactions, see *Oracle8 Tuning*.

16

Advanced Queuing

Many that are first shall be last; and the last shall be first.

Matthew 19:30: *The Bible*

This chapter describes the Oracle Advanced Queuing (Oracle AQ) feature. The chapter includes:

- Introduction to Message Queuing
- Oracle Advanced Queuing
  - Queuing Entities
  - Features of Advanced Queuing

Attention:

- If you purchase the product *Oracle8*, you cannot use Oracle AQ.
- If you purchase the product *Oracle8 Enterprise Edition* without the *Objects Option*, you can use Oracle AQ with queues of RAW datatype only.
- If you purchase the product *Oracle8 Enterprise Edition* with the *Objects Option*, you can use the full functionality of Oracle AQ.

See *Getting to Know Oracle8 and the Oracle8 Enterprise Edition* for more information about the features and options available with Oracle8 Enterprise Edition.

Additional Information: For more information about Oracle AQ, see *Oracle8 Application Developer's Guide*.

Introduction to Message Queuing

Communication between programs can be classified into one of two types:

- synchronous communication (online or connected model)
- asynchronous communication (disconnected or deferred model)

Synchronous Communication

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

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

The major drawback of the synchronous model of communication is that the programs must be available and running for the application to work. In the event of network or machine failure, programs cease to function.

Asynchronous Communication

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

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

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

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

Oracle Advanced Queuing

Oracle Advanced Queuing (Oracle AQ) integrates a message queuing system with the Oracle database. This allows you to store messages into queues for deferred retrieval and processing by the Oracle server.

Applications can access the queuing functionality through a PL/SQL interface. This provides a reliable and efficient queuing system without additional software like transaction processing (TP) monitors or Message Oriented Middleware.

Oracle AQ offers the following functionality:

- priority and ordering of queue elements
- ability to specify a window of execution for each queue element
- ability to query queues using standard SQL
- integrated transactions to simplify application development and management
- ability to dequeue multiple queue elements as a bundle
- ability to specify multiple recipients
- ability to propagate messages to queues in local or remote Oracle databases
- persistent queuing
- statistics on messages in queues

Oracle AQ queues are implemented in database tables, so that all the operational benefits of high availability, scalability, and reliability are applicable to queue data. In addition, database development and management tools can be used with queues.

Queuing Entities

Oracle AQ has five basic entities: messages, queues, queue tables, agents, and the queue monitor.

Messages

A *message* is the smallest unit of information being inserted into and retrieved from a queue. A message consists of control information and payload data. The control information represents message properties used by Oracle AQ to manage messages. The payload data is the information stored in the queue and is transparent to Oracle AQ. The datatype of the payload can be either RAW or an object type.

A message can reside in only one queue. A message is created by the enqueue call and consumed by the dequeue call. Enqueue and dequeue calls are part of the DBMS_AQ package.

Queues

A *queue* is the repository for messages. There are two types of queues: user (normal) queues and exception queues. The user queue is for normal message processing. All messages in a queue must have the same datatype. Messages are transfered to an exception queue if they cannot be retrieved and processed for some reason.

Queues can be created, altered, started, stopped, and dropped by using the DBMS_AQADM package.

Queue Tables

Queues are stored in *queue tables*. Each queue table is a database table and contains one or more queues. Each queue table contains a default exception queue.

Creating a queue table creates a database table with approximately 25 columns. These columns store Oracle AQ metadata and the user-defined payload.

A view and two indexes are created on the queue table. The view allows you to query the message data. The indexes are used to accelerate access to message data.

Agents

An agent is a queue user. There are two types of agents: producers who place messages in a queue (enqueuing) and consumers who retrieve messages (dequeuing). Any number of producers and consumers can access the queue at a given time.

An agent is identified by its name, address, and protocol. For an agent on a remote database, the only protocol currently supported is an Oracle database link, using an address of the form *queue_name@dblink*.

Queue Monitor

The queue monitor is an optional background process that monitors messages in the queue. It provides the mechanism for message expiration, retry, and delay (see “Windows of Execution” on page 16-5) and allows you to collect interval statistics (see “Queuing Statistics” on page 16-8).

The queue monitor process is different from most other Oracle background processes in that process failure does not cause the instance to fail.

The initialization parameter AQ_TM_PROCESS specifies creation of one or more queue monitor processes at instance startup.

Features of Advanced Queuing

This section describes the major features of Oracle Advanced Queuing.

Structured Payload

You can use object types to structure and manage the payload. (The RAW datatype can be used for unstructured payloads.)

Integrated Database Level Operational Support

Oracle AQ stores messages in tables. All standard database features such as recovery, restart, and Oracle Enterprise Manager are supported.

SQL Access

Messages are stored as database records. You can use SQL to access the message properties, the message history, and the payload. All available SQL technology, such as indexes, can be used to optimize the access to messages.

The AQ_ADMINISTRATOR role provides access to information about queues.

Windows of Execution

You can specify that the consumption of a message has to occur in a specific time window. A message can be marked as available for processing only after a specified time elapses (a delay time) and as having to be consumed before a specified time limit expires.

The initialization parameter AQ_TM_PROCESS enables time monitoring on queue messages, which is used for messages that specify delay and expiration properties. Time monitoring must also be enabled if you want to collect interval statistics (see "Queuing Statistics" on page 16-8).

If this parameter is set to 1, Oracle creates one *queue monitor process* (QMN0) as a background process to monitor the messages. If it is set to 2 through 10, Oracle creates that number of QMN*n* processes; if the parameter is not specified or is set to 0, then queue monitor processes are not created. The procedures in the DBMS_AQADM package for starting and stopping queue monitor operations are only valid if at least one queue monitor process was started with this parameter as part of instance startup.

Multiple Consumers per Message

A single message can be consumed by multiple consumers.

Navigation

You have several options for selecting a message from a queue. You can select the first message or, once you have selected a message and established a consistent-read snapshot, you can retrieve the next message based on the current snapshot. You will acquire a new consistent-read snapshot every time you select the first message from the queue.

You can also retrieve a specific message using the message's correlation identifier.

Ordering of Messages

You have three options for specifying the order in which messages are consumed: A sort order that specifies which properties are used to order all message in a queue, a priority that can be assigned to each message, and a sequence deviation that allows you to position a message in relation to other messages.

If several consumers act on the same queue, a consumer will get the first message that is available for immediate consumption. A message that is in the process of being consumed by another consumer will be skipped.

Modes of Dequeue

A DEQUEUE request can either browse or remove a message. If a message is browsed it remains available for further processing. If a message is removed, it is not available any more for DEQUEUE requests. Depending on the queue properties a removed message may be retained in the queue table.

Waiting for the Arrival of Messages

A DEQUEUE could be issued against an empty queue. You can specify if and for how long the request is allowed to wait for the arrival of a message.

Retries with Delays

A message has to be consumed exactly once. If an attempt to dequeue a message fails and the transaction is rolled back, the message will be made available for reprocessing after a user-specified delay elapses. Reprocessing will be attempted up to the specified limit.

Exception Queues

A message may not be consumed within the given constraints, that is, within the window of execution or within the limits of the retries. If such a condition arises, the message will be moved to a user-specified exception queue.

Visibility

ENQUEUE/DEQUEUE requests are normally part of a transaction that contains the requests. This provides the desired transactional behavior. However, you can specify that a request is a transaction by itself, making the result of that request immediately visible to other transactions.

Message Grouping

Messages belonging to one queue can be grouped to form a set that can only be consumed by one user at a time. This requires that the queue be created in a queue table that is enabled for message grouping.

All messages belonging to a group have to be created in the same transaction and all messages created in one transaction belong to the same group. This feature allows you to segment complex messages into simple messages, for example, messages directed to a queue containing invoices could be constructed as a group of messages starting with the header message, followed by messages representing details, followed by the trailer message.

Retention

You can specify that messages be retained after consumption. This allows you to keep a history of relevant messages. The history can be used for tracking, data warehouse, and data mining operations.

Message History

Oracle AQ stores information about the history of each message. The information contains the ENQUEUE/DEQUEUE time and the identification of the transaction that executed each request.

Tracking

If messages are retained they can be related to each other; for example, if a message *m2* is produced as a result of the consumption of message *m1*, *m1* is related to *m2*. This allows you to track sequences of related messages. These sequences represent "event journals" which are often constructed by applications. Oracle AQ is designed to let applications create event journals automatically.

Propagatimg Messages to Other Databases

Messages enqueued in one database can be propagated to queues on another database. The datatypes of the source and destination queues must match each other.

Message propagation enables applications to communicate with each other without being connected to the same database or the same queue. Propagation uses database links and Net8 between local or remote databases, both of which must have Oracle AQ enabled.

You can schedule (or unschedule) message propagation and specify the start time, the propagation window, and a date function for later propagation windows in periodic schedules. The data dictionary view DBA_QUEUE_SCHEDULES describes the current schedules for propagating messages.

The job queue background processes (SNP*n*) handle message propagation. To enable propagation, you must start at least one job queue process with the initialization parameter JOB_QUEUE_PROCESSES.

Queuing Statistics

Oracle AQ keeps statistics about the current state of the queuing system as well as time-interval statistics in the dynamic table GV$AQ.

Statistics about the current state of the queuing system include the numbers of ready, waiting, and expired messages.

One or more queue monitor processes must be started (see "Windows of Execution" on page 16-5) to keep interval statistics, which include:

- the number of messages in each state (ready, waiting, and expired)
- the average wait time of waiting messages
- the total wait time of waiting messages

Import/Export

The import / export of queues constitutes the import / export of the underlying queue tables and related dictionary tables. Import and export of queues can only be done at queue table granularity.

When a queue table is exported, both the table definition information and the queue data are exported. When a queue table is imported, export action procedures maintain the queue dictionary. Because the queue table data is also exported, the user is responsible for maintaining application-level data integrity when queue table data are being transported.

For every queue table that supports multiple recipients, there is an index-organized table that contains important queue metadata. This metadata is essential to the operations of the queue, so you must export and import this index-organized table as well as the queue table for the queues in this table to work after import.

When the schema containing the queue table is exported, the index-organized table is also automatically exported. The behavior is similar at import time. Because the metadata table contains ROWIDs of some rows in the queue table, import issues a note about the ROWIDs being obsolete when importing the metadata table. This message can be ignored, as the queuing system automatically corrects the obsolete ROWIDs as a part of the import process. However, if another problem is encountered while doing the import (such as running out of rollback segment space), the problem should be corrected and the import should be rerun.

Correlation Identifier

You can assign an identifier to each message. This identifier can be used to retrieve specific messages.

Oracle Enterprise Manager

Oracle Enterprise Manager provides a graphical interface (GUI) for some of the queue administration functions, including starting and stopping a queue, scheduling and unscheduling propagation, and viewing queue properties as part of the schema manager.

> **Additional Information:** For detailed information about Oracle AQ, see the *Oracle8 Application Developer's Guide.*

17

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

For information about the dependencies of procedures, functions, and packages, and how Oracle manages these dependencies, see Chapter 19, “Oracle Dependency Management”.

> **Additional Information:** If you are using Trusted Oracle, see your *Trusted Oracle* documentation.

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.

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 the Oracle server.

While many Oracle products have PL/SQL components, this chapter specifically covers the procedures and packages that can be stored in an Oracle database and processed using the Oracle server PL/SQL engine. The PL/SQL capabilities of each Oracle tool are described in the appropriate tool's documentation. For more information, see "PL/SQL" on page 14-15.

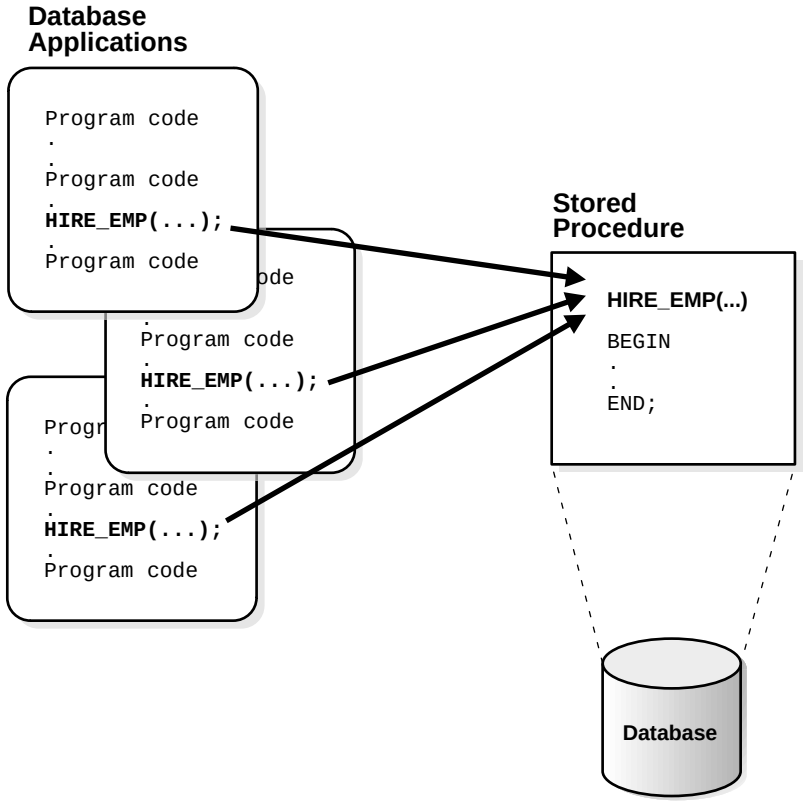
Stored Procedures and Functions

Procedures and functions are schema objects that logically group a set of SQL and other PL/SQL programming language statements together to perform a specific task. Procedures and functions are created in a user's schema and stored in a database for continued use. You can execute a procedure or function interactively 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 17–1 illustrates a simple procedure that is stored in the database and called by several different database applications.

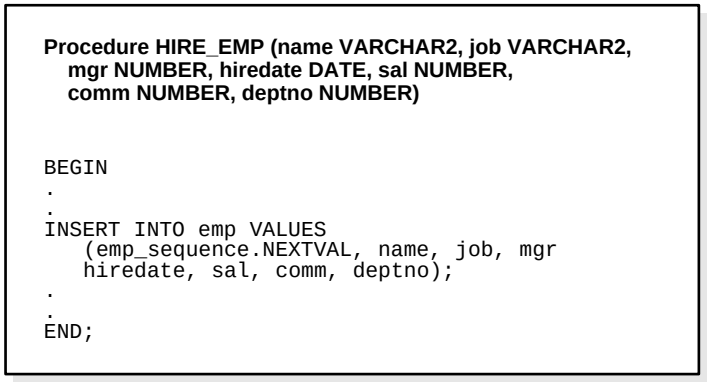
Procedures and functions are identical except that functions always return a single value to the caller, while procedures do not. For simplicity, the term "procedure" as used in the remainder of this chapter means "procedure or function".

Figure 17–1 A Stored Procedure



The stored procedure in Figure 17–1, which inserts an employee record into the EMP table, is shown in Figure 17–2.

Figure 17–2 The HIRE_EMP Procedure



All of the database applications in Figure 17–1 call the HIRE_EMP procedure. Alternatively, a privileged user might use Oracle Enterprise Manager or Server Manager to execute the HIRE_EMP procedure using the following statement:

```
EXECUTE hire_emp ('TSMITH', 'CLERK', 1037, SYSDATE, \
                    500, NULL, 20);
```

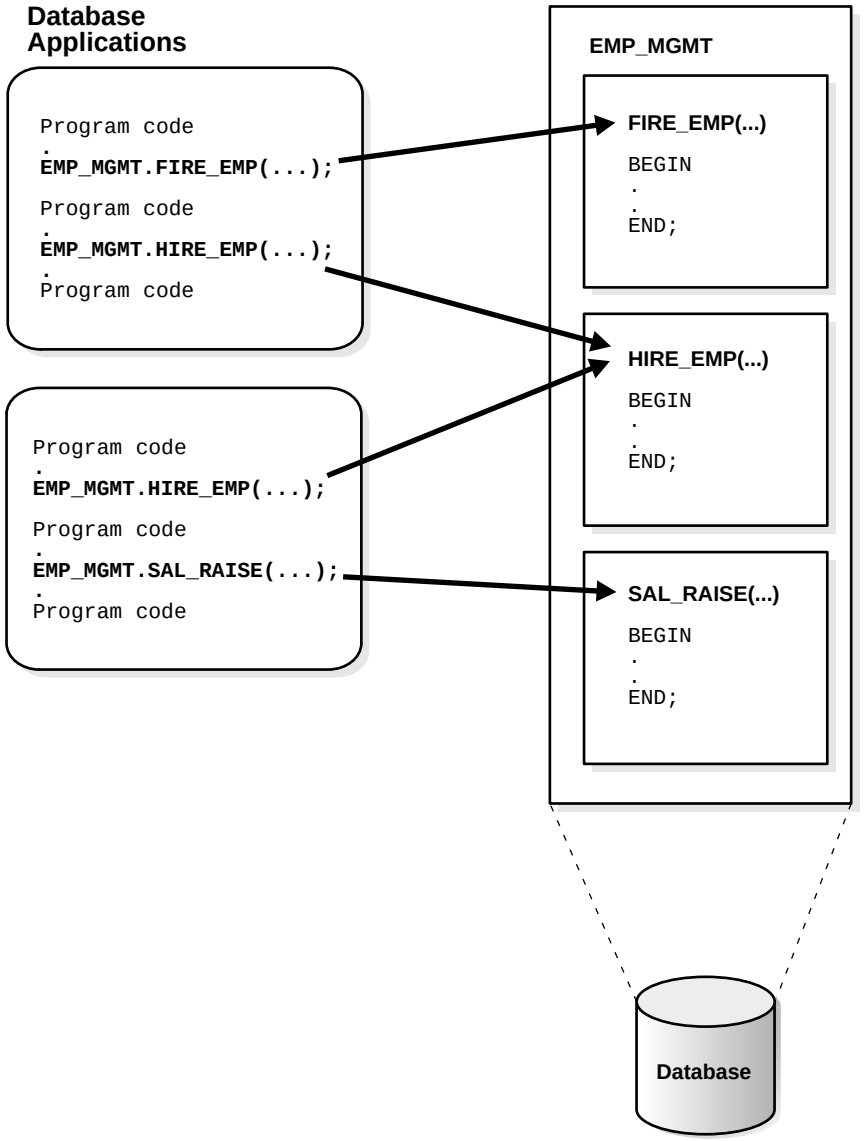
This statement places a new employee record for TSMITH in the EMP table.

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 17–3 illustrates a package that encapsulates a number of procedures used to manage an employee database.

Figure 17–3 A Stored Package



Database applications explicitly call packaged procedures as necessary. After being granted the privileges for the EMP_MGMT package, a user can explicitly execute any of the procedures contained in it. For example, the following statement might be issued using Oracle Enterprise Manager or 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 (see "Packages" on page 17-10).

Procedures and Functions

A *procedure* or *function* is a schema object that consists of a set of SQL statements and other PL/SQL constructs, grouped together, stored in the database, and executed as a unit to solve a specific problem or perform a set of related tasks. Procedures and functions permit the caller to provide parameters that can be input only, output only, or input and output values. Procedures and functions allow you to combine the ease and flexibility of SQL with the procedural functionality of a structured programming language.

For example, the following statement creates the CREDIT_ACCOUNT procedure, which credits money to a bank account:

```
CREATE PROCEDURE credit_account
      (acct NUMBER, credit NUMBER) AS
/* This procedure accepts two arguments: an account
   number and an amount of money to credit to the specified
   account. If the specified account does not exist, a
   new account is created. */

        old_balance  NUMBER;
        new_balance  NUMBER;
   BEGIN
        SELECT balance INTO old_balance FROM accounts
          WHERE acct_id = acct
          FOR UPDATE OF balance;

        new_balance := old_balance + credit;
        UPDATE accounts SET balance = new_balance
          WHERE acct_id = acct;
        COMMIT;
```

```
        EXCEPTION
          WHEN NO_DATA_FOUND THEN
            INSERT INTO accounts (acct_id, balance)
                VALUES(acct, credit);
          WHEN OTHERS THEN
        ROLLBACK;
END credit_account;
```

Notice that this sample procedure includes both SQL and PL/SQL statements.

Procedure Guidelines

Use the following guidelines to design and use all stored procedures:

- Define procedures to complete a single, focused task. Do not define long procedures with several distinct subtasks, because subtasks common to many procedures might be duplicated unnecessarily in the code of several procedures.
- Do not define procedures that duplicate the functionality already provided by other features of Oracle. For example, do not define procedures to enforce simple data integrity rules that you could easily enforce using declarative integrity constraints.

Benefits of Procedures

Procedures provide advantages in the following areas.

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 who have only the privilege to execute the procedure (but not the privileges to query, update, or delete from the underlying tables) can invoke the procedure, but they cannot manipulate table data in any other way.

Performance

Stored procedures can improve database performance in several ways:

- The amount of information that must be sent over a network is small compared to issuing individual SQL statements or sending the text of an entire PL/SQL

block to Oracle, because the information is sent only once and thereafter invoked when it is used.

- A procedure's compiled form is readily available in the database, so no compilation is required at execution time.
- If the procedure is already present in the shared pool of the SGA, retrieval from disk is not required, and execution can begin immediately.

Memory Allocation

Because stored procedures take advantage of the shared memory capabilities of Oracle, only a single copy of the procedure needs to be loaded into memory for execution by multiple users. Sharing the same code among many users results in a substantial reduction in Oracle memory requirements for applications.

Productivity

Stored procedures increase development productivity. By designing applications around a common set of procedures, you can avoid redundant coding and increase your productivity.

For example, procedures can be written to insert, update, or delete 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

A stored procedure is created and stored in the database as a schema object. Once created and compiled, it is a named object that can be executed without recompil-

ing. Additionally, dependency information is stored in the data dictionary to guarantee the validity of each stored procedure.

As an alternative to a stored procedure, you can create an anonymous PL/SQL block by sending an unnamed PL/SQL block to the Oracle server from an Oracle tool or an application. Oracle compiles the PL/SQL block and places the compiled version in the shared pool of the SGA, but does not store the source code or compiled version in the database for reuse beyond the current instance. Shared SQL allows anonymous PL/SQL blocks in the shared pool to be reused and shared until they are flushed out of the shared pool.

In either case, moving PL/SQL blocks out of a database application and into database procedures stored either in the database or in memory, you avoid unnecessary procedure recompilations by Oracle at runtime, improving the overall performance of the application and Oracle.

Standalone Procedures

Stored procedures not defined within the context of a package are called *standalone procedures*. Procedures defined within a package are considered a part of the package. (See "Packages" on page 17-10 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 19, "Oracle Dependency Management", for more information about dependency tracking.

External Procedures

A PL/SQL procedure executing on an Oracle server can call an external procedure or function that is written in the C programming language and stored in a shared library. The C routine executes in a separate address space from that of the Oracle server.

> **Additional Information:** See *PL/SQL User's Guide and Reference* for information about external procedures.

Packages

Packages encapsulate related procedures, functions, and associated cursors and variables together as a unit in the database.

You create a package in two parts: the specification and the body. A package's *specification* declares all public constructs of the package and the *body* defines all constructs (public and private) of the package. This separation of the two parts provides the following advantages:

- The developer has more flexibility in the development cycle. You can create specifications and reference public procedures without actually creating the package body.
- You can alter procedure bodies contained within the package body separately from their publicly declared specifications in the package specification. As long as the procedure specification does not change, objects that reference the altered procedures of the package are never marked invalid; that is, they are never marked as needing recompilation. (For more information about dependencies, see Chapter 19, “Oracle Dependency Management”.)

The following example creates the specification and body for a package that contains several procedures and functions that process banking transactions.

```
CREATE PACKAGE bank_transactions (null) AS
  minimum_balance  CONSTANT NUMBER := 100.00;
  PROCEDURE apply_transactions;
  PROCEDURE enter_transaction (acct   NUMBER,
                               kind   CHAR,
                               amount NUMBER);
END bank_transactions;

CREATE PACKAGE BODY bank_transactions AS

/*  Package to input bank transactions */

   new_status  CHAR(20);  /* Global variable to record status
                             of transaction being applied. Used
                             for update in APPLY_TRANSACTIONS. */

   PROCEDURE do_journal_entry (acct NUMBER,
                               kind CHAR) IS

/*  Records a journal entry for each bank transaction applied
    by the APPLY_TRANSACTIONS procedure. */
```

```
  BEGIN
        INSERT INTO journal
          VALUES (acct, kind, sysdate);
        IF kind = 'D' THEN
          new_status := 'Debit applied';
        ELSIF kind = 'C' THEN
          new_status := 'Credit applied';
        ELSE
          new_status := 'New account';
        END IF;
  END do_journal_entry;

  PROCEDURE credit_account (acct NUMBER, credit NUMBER) IS

/* Credits a bank account the specified amount. If the account
   does not exist, the procedure creates a new account first. */

        old_balance  NUMBER;
        new_balance  NUMBER;

  BEGIN
        SELECT balance INTO old_balance FROM accounts
          WHERE acct_id = acct
          FOR UPDATE OF balance; /* Locks account for credit update */

        new_balance := old_balance + credit;
        UPDATE accounts SET balance = new_balance
          WHERE acct_id = acct;
        do_journal_entry(acct, 'C');

  EXCEPTION
        WHEN NO_DATA_FOUND THEN  /* Create new account if not found */
          INSERT INTO accounts (acct_id, balance)
            VALUES(acct, credit);
          do_journal_entry(acct, 'N');
        WHEN OTHERS THEN  /* Return other errors to application */
          new_status := 'Error: ' || SQLERRM(SQLCODE);
  END credit_account;

  PROCEDURE debit_account (acct  NUMBER, debit NUMBER) IS

/* Debits an existing account if result is greater than the
   allowed minimum balance. */

        old_balance         NUMBER;
```

```
        new_balance         NUMBER;
        insufficient_funds  EXCEPTION;

    BEGIN
        SELECT balance INTO old_balance FROM accounts
          WHERE acct_id = acct
          FOR UPDATE OF balance;
        new_balance := old_balance - debit;
        IF new_balance >= minimum_balance THEN
          UPDATE accounts SET balance = new_balance

            WHERE acct_id = acct;
        do_journal_entry(acct, 'D');
        ELSE
          RAISE insufficient_funds;
        END IF;

    EXCEPTION
        WHEN NO_DATA_FOUND THEN
          new_status := 'Nonexistent account';
        WHEN insufficient_funds THEN
          new_status := 'Insufficient funds';
        WHEN OTHERS THEN  /* Returns other errors to application */
          new_status := 'Error: ' || SQLERRM(SQLCODE);
    END debit_account;

    PROCEDURE apply_transactions IS

  /* Applies pending transactions in the table TRANSACTIONS to the
     ACCOUNTS table. Used at regular intervals to update bank
     accounts without interfering with input of new transactions. */

  /* Cursor fetches and locks all rows from the TRANSACTIONS
     table with a status of 'Pending'. Locks released after all
           pending transactions have been applied. */

        CURSOR trans_cursor IS
          SELECT acct_id, kind, amount FROM transactions
            WHERE status = 'Pending'
            ORDER BY time_tag
            FOR UPDATE OF status;

     BEGIN
        FOR trans IN trans_cursor LOOP   /* implicit open and fetch */
          IF trans.kind = 'D' THEN
```

```
            debit_account(trans.acct_id, trans.amount);
          ELSIF trans.kind = 'C' THEN
            credit_account(trans.acct_id, trans.amount);
          ELSE
            new_status := 'Rejected';
          END IF;
          /* Update TRANSACTIONS table to return result of applying
             this transaction. */
          UPDATE transactions SET status = new_status
            WHERE CURRENT OF trans_cursor;
        END LOOP;
        COMMIT;  /* Release row locks in TRANSACTIONS table. */
  END apply_transactions;
  PROCEDURE enter_transaction (acct   NUMBER,
                               kind   CHAR,
                               amount NUMBER) IS

/* Enters a bank transaction into the TRANSACTIONS table. A new
   transaction is always put into this 'queue' before being
   applied to the specified account by the APPLY_TRANSACTIONS
   procedure. Therefore, many transactions can be simultaneously
   input without interference. */

  BEGIN
        INSERT INTO transactions
          VALUES (acct, kind, amount, 'Pending', sysdate);
        COMMIT;
  END enter_transaction;

END bank_transactions;
```

Packages allow the database administrator or application developer to organize similar routines. They also offer increased functionality and database performance.

Benefits of Packages

Packages are used to define related procedures, variables, and cursors and are often implemented to provide advantages in the following areas:

- encapsulation of related procedures and variables
- declaration of public and private procedures, variables, constants, and cursors
- better performance

Encapsulation

Stored packages allow you to encapsulate (group) related stored procedures, variables, datatypes, and so forth 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. You can define the package so that only three procedures are public and therefore available for execution by a user of the package; the remainder of the procedures are private and can only be accessed by the procedures within the package.

Do not confuse public and private package variables with grants to PUBLIC, which are described in Chapter 25, “Controlling Database Access”.

Performance Improvement

An entire package is loaded into memory when a procedure within the package is called for the first time. This load is completed in one operation, as opposed to the separate loads required for standalone procedures. Therefore, when calls to related packaged procedures occur, no disk I/O is necessary to execute the compiled code already in memory.

A package body can be replaced and recompiled without affecting the specification. As a result, schema objects that reference a package's constructs (always via the specification) need not be recompiled unless the package specification is also replaced. By using packages, unnecessary recompilations can be minimized, resulting in less impact on overall database performance.

Dependency Tracking for Packages

A package is dependent on the objects referenced by the procedures and functions defined in its body. Oracle automatically tracks and manages such dependencies.

See Chapter 19, “Oracle Dependency Management”, for more information about dependency tracking.

How Oracle Stores Procedures and Packages

When you create a procedure or package, Oracle

- compiles the procedure or package
- stores the compiled code in memory
- stores the procedure or package in the database

Compiling Procedures and Packages

The PL/SQL compiler compiles the source code. The PL/SQL compiler is part of the PL/SQL engine contained in Oracle. If an error occurs during compilation, a message is returned.

> **Additional Information:** Information on identifying compilation errors is contained in the *Oracle8 Application Developer’s Guide*.

Storing the Compiled Code in Memory

Oracle caches the compiled procedure or package in the shared pool of the system global area (SGA). This allows the code to be executed quickly and shared among many users. The compiled version of the procedure or package remains in the shared pool according to the modified least-recently-used algorithm used by the shared pool, even if the original caller of the procedure terminates his or her session. See “The Shared Pool” on page 6-6 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:

schema object name	This name identifies the procedure or package. You specify this name in the CREATE PROCEDURE, CREATE FUNCTION, CREATE PACKAGE, or CREATE PACKAGE BODY statement.
source code and parse tree	The PL/SQL compiler parses the source code and produces a parsed representation of the source code, called a *parse tree*.

pseudocode (P code)	The PL/SQL compiler generates the *pseudocode*, or P code, based on the parsed code. The PL/SQL engine executes this when the procedure or package is invoked.
error messages	Oracle might generate errors during the compilation of a procedure or package.

To avoid unnecessary recompilation of a procedure or package, both the parse tree and the P code of an object are stored in the database. This allows the PL/SQL engine to read the compiled version of a procedure or package into the shared pool buffer of the SGA when it is invoked and not currently in the SGA. The parse tree is used when the code calling the procedure is compiled.

All parts of database procedures are stored in the data dictionary (which is in the SYSTEM tablespace) of the corresponding database. When planning the size of the SYSTEM tablespace, the database administrator should keep in mind that all stored procedures require space in this tablespace.

How Oracle Executes Procedures and Packages

When you invoke a standalone or packaged procedure, Oracle verifies user access, verifies procedure validity, and 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 determine whether the status of the procedure or package is valid or invalid. A procedure or package is invalid when one of the following has occurred since the procedure or package was last compiled:

- One or more of the schema objects referenced within the procedure or package (such as tables, views, and other procedures) have been altered or dropped (for example, 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 schema object privilege for one or more of the schema objects referenced by a procedure or package has been revoked from PUBLIC or from the owner of the procedure or package.

A procedure is *valid* if it has not been invalidated by any of the above operations. If a valid standalone or packaged procedure is called, 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 19, "Oracle Dependency Management".

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 14–2 on on page 14-16.

18

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
- Types of Triggers
- Trigger Execution

> **Additional Information:** If you are using Trusted Oracle, see your *Trusted Oracle* documentation.

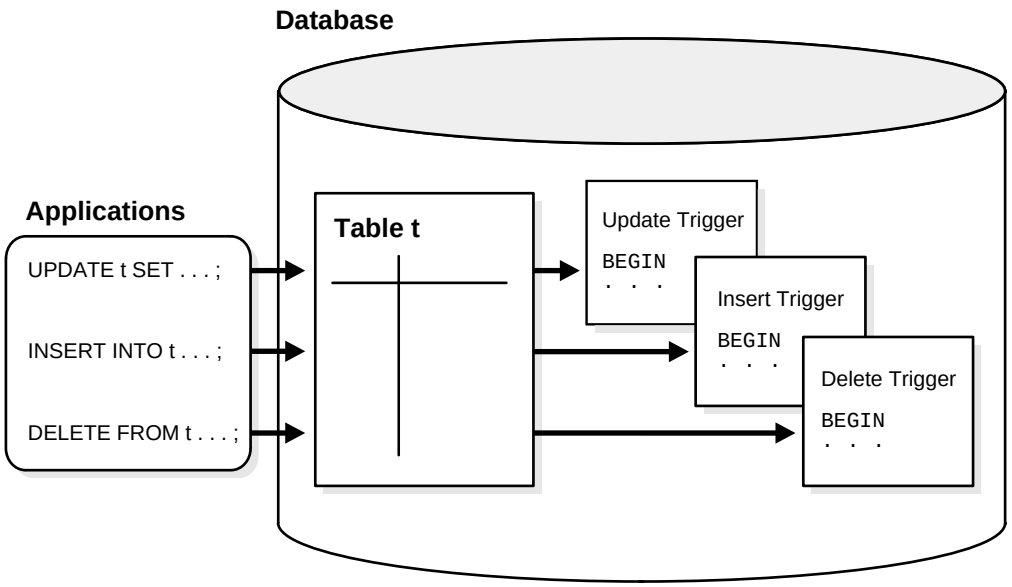
An Introduction to Triggers

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 17, “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. A procedure is explicitly executed by a user, application, or trigger. Triggers (one or more) 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.

Figure 18–1 shows a database application with some SQL statements that implicitly fire several triggers stored in the database.

Figure 18–1 Triggers



Notice that triggers are stored in the database separate 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

Triggers can supplement the standard capabilities of Oracle to provide a highly customized database management system. For example, a trigger can restrict DML operations against a table to those issued during regular business hours. A trigger could also restrict DML operations to occur only at certain times during weekdays. Other uses for triggers are 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

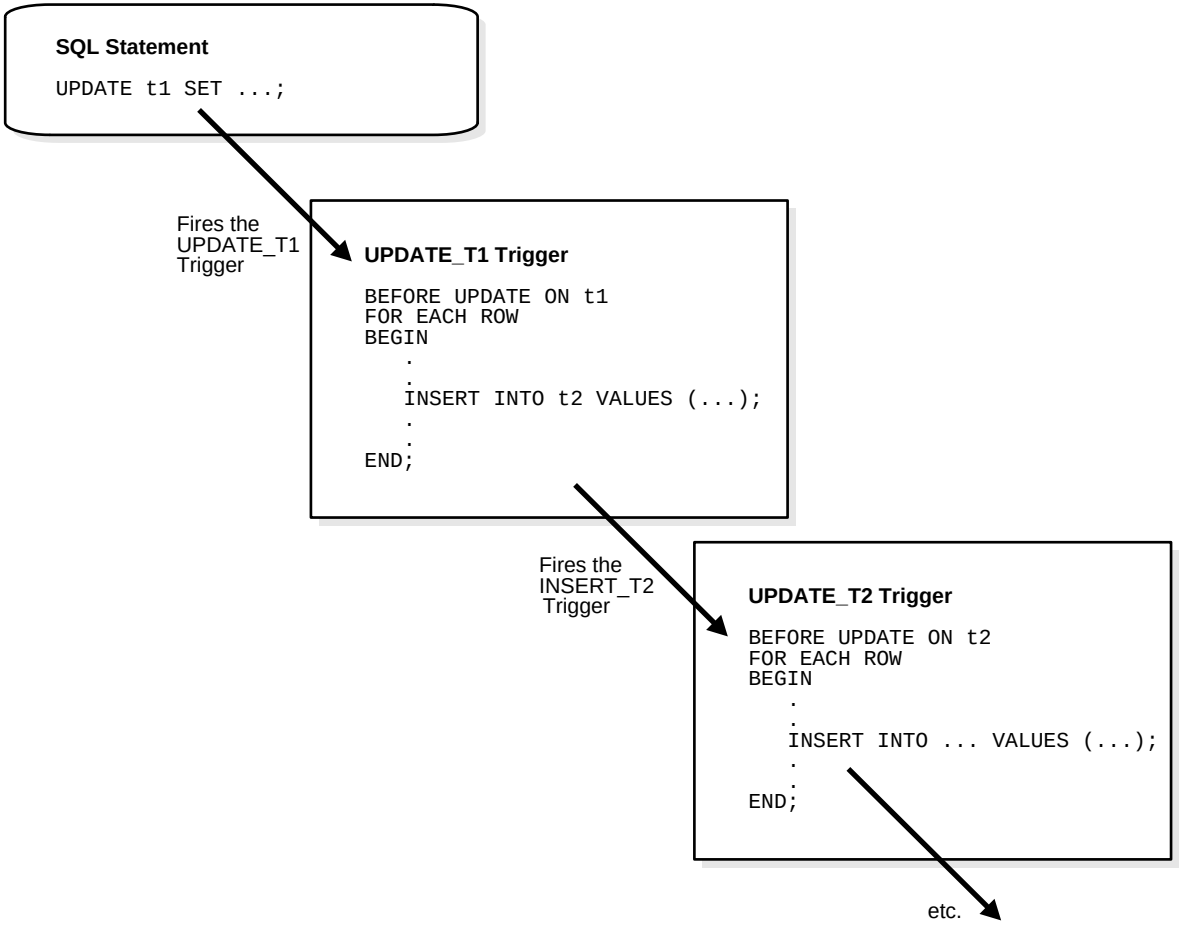
> **Additional Information:** Examples of many of these trigger uses are included in the *Oracle8 Application Developer's Guide*.

Some Cautionary Notes about Triggers

Triggers are useful for customizing a database. However, you should use triggers only when necessary. The excessive use of triggers can result in complex interdependencies, which may be difficult to maintain in a large application. For example, when a trigger is fired, a SQL statement within its trigger action potentially can fire other triggers, as illustrated in Figure 18–2.

When a statement in a trigger body causes another trigger to be fired, the triggers are said to be *cascading*.

Figure 18–2 Cascading Triggers



Note: Oracle Forms can define, store, and execute triggers of a different sort. However, do not confuse Oracle Forms triggers with the database triggers discussed in this chapter.

Triggers vs. Declarative Integrity Constraints

You can use both database triggers and integrity constraints to define and enforce any type of integrity rule. However, Oracle Corporation strongly recommends that you use database triggers to constrain data input only in the following situations:

- when a required referential integrity rule cannot be enforced using the following integrity constraints:
  - NOT NULL, UNIQUE key
  - PRIMARY KEY
  - FOREIGN KEY
  - CHECK
  - 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 about integrity constraints, see “How Oracle Enforces Data Integrity” on page 24-4.

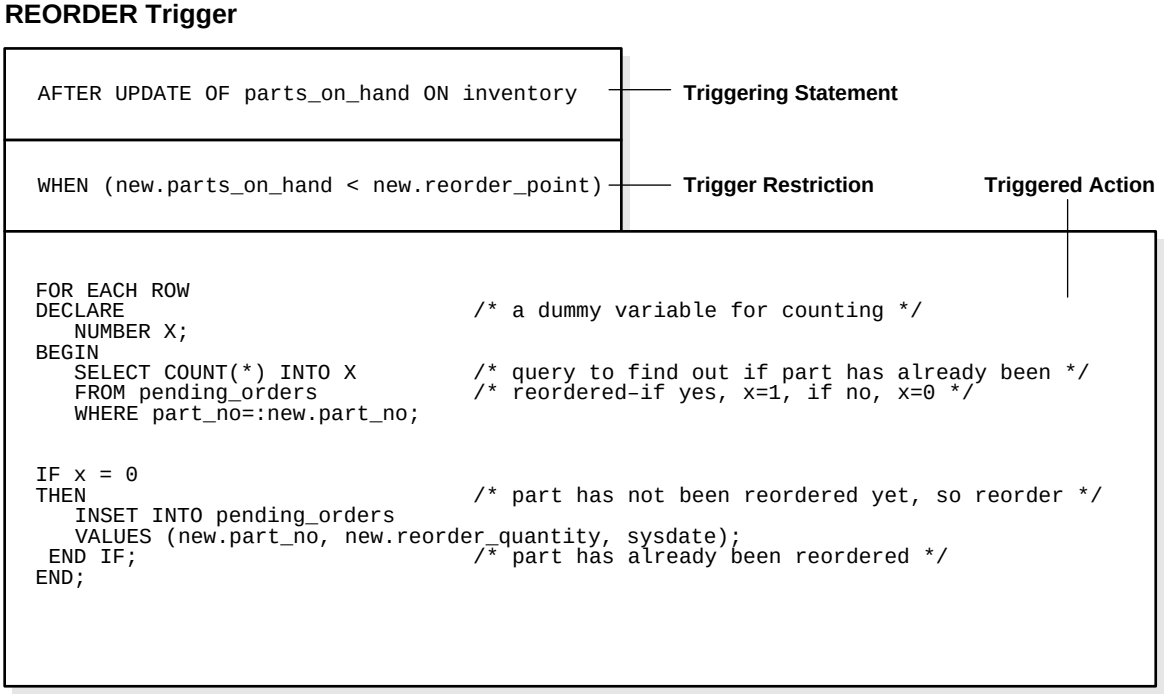
Parts of a Trigger

A trigger has three basic parts:

- a triggering event or statement
- a trigger restriction
- a trigger action

Figure 18–3 represents each of these parts of a trigger and is not meant to show exact syntax. The sections that follow explain each part of a trigger in greater detail.

Figure 18–3 The REORDER Trigger



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 18–3, the triggering statement is

```
. . . UPDATE OF parts_on_hand ON inventory . . .
```

which means: 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. You cannot specify a column list for INSERT and DELETE statements, because they affect entire rows of information.

A triggering event can specify multiple DML statements, as in

```
. . . INSERT OR UPDATE OR DELETE OF inventory . . .
```

which means: 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, you can use conditional predicates to detect the type of triggering statement. In this way, you can create a single trigger that executes different code based on the type of statement that fires the trigger.

Trigger Restriction

A trigger restriction specifies a Boolean (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. In the example, the trigger restriction is

```
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.

Like 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 triggers (described in the next section), 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

This section describes the different types of triggers:

- row and statement triggers
- BEFORE and AFTER triggers
- INSTEAD OF triggers

Row Triggers and Statement Triggers

When you define a trigger, you can specify the number of times the trigger action is to be executed: once for every row affected by the triggering statement (such as 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 18–3 illustrates a row trigger that uses the values of each row affected by the triggering statement.

Statement Triggers

A *statement trigger* is fired once on behalf of the triggering statement, regardless of the number of rows in the table that the triggering statement affects (even if no rows are affected). For example, if a DELETE statement deletes several rows from a table, a statement-level DELETE trigger is fired only once, 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 and AFTER Triggers

When defining a trigger, you can specify the *trigger timing*—whether the trigger action is to be executed before or after the triggering statement. BEFORE and AFTER apply to both statement and row triggers. (Another type of trigger is described in “INSTEAD OF Triggers” on page 18-11.)

BEFORE Triggers

BEFORE triggers execute the trigger action before the triggering statement is executed. This type of trigger is commonly used in the following situations:

- When the trigger action should determine whether the triggering statement should be allowed to complete. Using a BEFORE trigger for this purpose, you can eliminate unnecessary processing of the triggering statement and its eventual rollback in cases where an exception is raised in the trigger action.
- To derive specific column values before completing a triggering INSERT or UPDATE statement.

AFTER Triggers

AFTER triggers execute the trigger action after the triggering statement is executed. AFTER triggers are used in the following situations:

- 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.

Trigger Combinations

Using the options listed above, 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. The example below con-

tains 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. Finally the statistical information is saved in the table STAT_TAB by the AFTER statement trigger.

Sample Package and Trigger for SAL Table

```
DROP TABLE stat_tab;
CREATE TABLE stat_tab(utype CHAR(8),
      rowcnt INTEGER, uhour INTEGER);

CREATE OR REPLACE PACKAGE stat IS
 rowcnt INTEGER;
END;
/

CREATE TRIGGER bt BEFORE UPDATE OR DELETE OR INSERT ON sal
BEGIN
 stat.rowcnt := 0;
END;
/

CREATE TRIGGER rt BEFORE UPDATE OR DELETE OR INSERT ON sal
FOR EACH ROW BEGIN
 stat.rowcnt := stat.rowcnt + 1;
END;
/

CREATE TRIGGER at AFTER UPDATE OR DELETE OR INSERT ON sal
DECLARE
 typ  CHAR(8);
 hour NUMBER;
BEGIN
 IF updating
 THEN typ := 'update'; END IF;
 IF deleting  THEN typ := 'delete'; END IF;
 IF inserting THEN typ := 'insert'; END IF;

 hour := TRUNC((SYSDATE - TRUNC(SYSDATE)) * 24);
 UPDATE stat_tab
    SET rowcnt = rowcnt + stat.rowcnt
   WHERE utype = typ
     AND uhour = hour;
 IF SQL%ROWCOUNT = 0 THEN
```

```
   INSERT INTO stat_tab VALUES (typ, stat.rowcnt, hour);
 END IF;

EXCEPTION
 WHEN dup_val_on_index THEN
   UPDATE stat_tab
      SET rowcnt = rowcnt + stat.rowcnt
    WHERE utype = typ
      AND uhour = hour;
END;
/
```

INSTEAD OF Triggers

INSTEAD OF triggers provide a transparent way of modifying views that cannot be modified directly through SQL DML statements (INSERT, UPDATE, and DELETE). These triggers are called INSTEAD OF triggers because, unlike other types of triggers, Oracle fires the trigger instead of executing the triggering statement. The trigger performs update, insert, or delete operations directly on the underlying tables.

You can write normal INSERT, DELETE, and UPDATE statements against the view and the INSTEAD OF trigger works invisibly in the background to make the right actions take place. By default, INSTEAD OF triggers are activated for each row.

Modifying Views

Modifying views has inherent problems of ambiguity.

- Deleting a row in a view could either mean deleting it from the base table or updating some column values so that it will no longer be selected by the view.
- Inserting a row in a view could either mean inserting a new row into the base table or updating an existing row so that it will be projected by the view.
- Updating a column in a view that involves joins might change the semantics of other columns that are not projected by the view.

Object views present additional problems (see Chapter 13, “Object Views”). For example, a key use of object views is to represent master / detail relationships. This inevitably involves joins, but modifying joins is inherently ambiguous.

As a result of these ambiguities, there are many restrictions on which views are modifiable (see the next section). An INSTEAD OF trigger can be used on object views as well as relational views that are not otherwise modifiable.

The mechanism of INSTEAD OF triggers also enables you to modify object view instances on the client-side through OCI. To modify an object materialized by an object view in the client-side object cache and flush it back to the persistent store, you must specify INSTEAD OF triggers, unless the object view is modifiable. If the object is read only, however, it is not necessary to define triggers to pin it.

> **Additional Information:** See *Oracle Call Interface Programmer's Guide* for more information.

Views That Are Not Modifiable

A view is *inherently modifiable* if it can be inserted, updated, or deleted without using INSTEAD OF triggers and if it conforms to the restrictions listed below. If the view query contains any of the following constructs, the view is not inherently modifiable and you therefore cannot perform inserts, updates, or deletes on the view:

- set operators
- group functions
- GROUP BY, CONNECT BY, or START WITH clauses
- the DISTINCT operator
- joins (however, a subset of join views are updatable — see "Updatable Join Views" on page 8-13)

If a view contains pseudocolumns or expressions, you can only update the view with an UPDATE statement that does not refer to any of the pseudocolumns or expressions.

Example of an INSTEAD OF Trigger

The following example shows an INSTEAD OF trigger for inserting rows into the MANAGER_INFO view.

```
CREATE VIEW manager_info AS
  SELECT e.name, e.empno, d.dept_type, d.deptno, p.level,
         p.projno
    FROM   emp e, dept d, project p
    WHERE  e.empno =  d.mgr_no
    AND    d.deptno = p.resp_dept;

CREATE TRIGGER manager_info_insert
  INSTEAD OF INSERT ON manager_info
  REFERENCING NEW AS n              -- new manager information
```

```
FOR EACH ROW
BEGIN
  IF NOT EXISTS SELECT * FROM emp
        WHERE emp.empno = :n.empno
  THEN
        INSERT INTO emp
            VALUES(:n.empno, :n.name);
  ELSE
        UPDATE emp SET emp.name = :n.name
            WHERE emp.empno = :n.empno;
    END IF;

    IF NOT EXISTS SELECT * FROM dept
        WHERE dept.deptno = :n.deptno
    THEN
        INSERT INTO dept
            VALUES(:n.deptno, :n.dept_type);
    ELSE
        UPDATE dept SET dept.dept_type = :n.dept_type
            WHERE dept.deptno = :n.deptno;
    END IF;

    IF NOT EXISTS SELECT * FROM project
        WHERE project.projno = :n.projno
    THEN
        INSERT INTO project
            VALUES(:n.projno, :n.project_level);
    ELSE
        UPDATE project SET project.level = :n.level
            WHERE project.projno = :n.projno;
    END IF;
END;
```

The actions shown for rows being inserted into the MANAGER_INFO view first test to see if appropriate rows already exist in the base tables from which MANAGER_INFO is derived. The actions then insert new rows or update existing rows, as appropriate. Similar triggers can specify appropriate actions for UPDATE and DELETE.

> **Additional Information:** See the CREATE TRIGGER command in *Oracle8 SQL Reference* for more information about INSTEAD OF triggers.

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 schema objects referenced in the code of the trigger action
- uses two-phase commit if a trigger updates remote tables in a distributed database
- fires multiple triggers in an unspecified order, if more than one trigger of the same type exists for a given statement

The Execution Model for Triggers and Integrity Constraint Checking

A single SQL statement can potentially fire up to four types of triggers: BEFORE row triggers, BEFORE statement triggers, AFTER row triggers, 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 checked on tables changed by AFTER statement triggers.

        6. Statements of BEFORE row triggers executed.

        7. Integrity constraint checked on tables changed by BEFORE row triggers.

    8. SQL statement executed.

    9. Integrity constraint from SQL statement checked.

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 so they do not rely on the firing order of multiple triggers of the same type.

Data Access for Triggers

When a trigger is fired, the tables referenced in the trigger action might be currently undergoing changes by SQL statements in other users' transactions. In all cases, the SQL statements executed within triggers follow the common rules used for standalone SQL statements. In particular, if an uncommitted transaction has modified values that a trigger being fired either needs to read (query) or write (update), the SQL statements in the body of the trigger being fired use the following guidelines:

- Queries see the current read-consistent snapshot of referenced tables and any data changed within the same transaction.
- Updates wait for existing data locks to be released before proceeding.

The following examples illustrate these points.

Example: 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 of Triggers

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 the trigger is flushed from the shared pool.

For more information about compiling and storing PL/SQL code, see "How Oracle Stores Procedures and Packages" on page 17-15.

Execution of Triggers

Oracle executes a trigger internally using the same steps used for procedure execution. The only subtle difference is that a user has the right to fire a trigger if he or

she has the privilege to execute the triggering statement. Other than this, triggers are validated and executed the same way as stored procedures.

For more information, see “How Oracle Executes Procedures and Packages” on page 17-16.

Dependency Maintenance for Triggers

Like procedures, triggers are dependent on referenced objects. Oracle automatically manages the dependencies of a trigger on the schema objects referenced in its trigger action. The dependency issues for triggers are the same as those for stored procedures. Triggers are treated like stored procedures; they are inserted into the data dictionary.

For more information, see Chapter 19, “Oracle Dependency Management”.

19

Oracle Dependency Management

Whoever you are — I have always depended on the kindness of strangers.

Tennessee Williams: *A Streetcar Named Desire*

The definitions of some objects, including views and procedures, reference other objects, such as tables. As a result, the objects being defined are dependent on the objects referenced in their definitions. This chapter discusses the dependencies among schema objects and how Oracle automatically tracks and manages these dependencies. It includes:

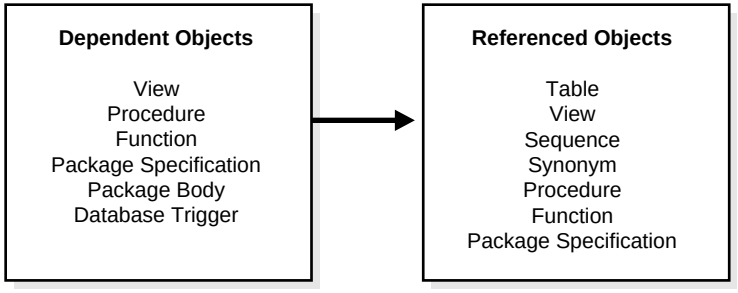
- An Introduction to Dependency Issues
- Resolving Schema Object Dependencies
- Dependency Management and Nonexistent Schema Objects
- Shared SQL Dependency Management
- Local and Remote Dependency Management

> **Additional Information:** If you are using Trusted Oracle, see your *Trusted Oracle* documentation 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 19–1 illustrates the different types of dependent and referenced objects.

Figure 19–1 Types of Possible Dependent and Referenced Schema Objects



If you alter the definition of a referenced object, dependent objects may or may not continue to function without error, depending on the type of alteration. For example, if you drop a table, no view based on the dropped table can be used.

Oracle automatically records dependencies among objects to alleviate the complex job of dependency management for the database administrator and users. For example, if you alter a table on which several stored procedures depend, Oracle automatically recompiles the dependent procedures the next time the procedures are referenced (executed or compiled against).

To manage dependencies among schema objects, all of the schema objects in a database have a status:

VALID	The object has been compiled and can be immediately used when referenced.

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.

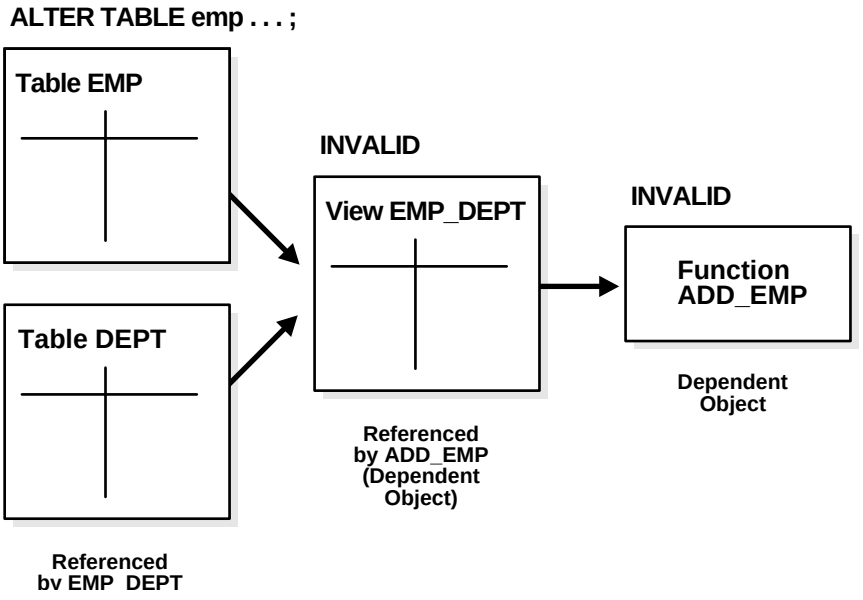
If a view, procedure, function, or package is invalid, Oracle may have attempted to compile it, but errors relating to the object occurred. For example, when compiling a view, one of its base tables might not exist, or the correct privileges for the base table might not be present. When compiling a package, there might be a PL/SQL or SQL syntax error, or the correct privileges for a referenced object might not be present. Objects with such problems remain invalid.

Oracle automatically tracks specific changes in the database and records the appropriate status for related objects in the data dictionary.

Status recording is a recursive process; any change in the status of a referenced object not only changes the status for directly dependent objects, but also for indirectly dependent objects.

For example, consider a stored procedure that directly references a view. In effect, the stored procedure indirectly references the base table(s) of that view. Therefore, if you alter a base table, the view is invalidated, which then invalidates the stored procedure. Figure 19–2 illustrates this.

Figure 19–2 Indirect Dependencies



Resolving Schema Object Dependencies

When a schema 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. Oracle's action depends on the status of the objects that are directly and indirectly referenced in a SQL statement:

- If every referenced object is valid, Oracle executes 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 invalid referenced objects 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 satisfied:

- The definition of the view or program unit must be correct; all SQL and PL/SQL statements must be proper constructs.
- All referenced objects must be present and of the expected structure. For example, if the defining query of a view includes a column, the column must be present in the base table.
- The **owner** of the view or program unit must have the necessary privileges for the referenced objects. For example, if a SQL statement in a procedure inserts a row into a table, the owner of the procedure must have the INSERT privilege for the referenced table.

Views and Base Tables

A view depends on the base tables (or views) referenced in its defining query. If the defining query of a view is not explicit about which columns are referenced, for example, `SELECT * FROM table`, 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.

An attempt to use an invalid view automatically causes Oracle to recompile the view dynamically. After replacing the view, the view might be valid or invalid, depending on the following condition:

- All base tables referenced by the defining query of a view must exist. If a base table of a view is renamed or dropped, the view is invalidated and cannot be used. References to invalid views cause the referencing statement to fail. The view can be compiled only if the base table is renamed to its original name or the base table is recreated.

- 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 table` query, because the defining query is expanded at view creation time and permanently stored in the data dictionary.

Program Units and Referenced Objects

Oracle automatically invalidates a program unit when the definition of a referenced object is altered. For example, assume that a standalone procedure includes several statements that reference a table, a view, another standalone procedure, and a public package procedure. In that case, the following conditions hold:

- If the referenced table is altered, 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 by using packages.

Session State and Referenced Packages

Each session that references a package construct has its own instance of that package, including a persistent state of any public and private variables, cursors, and constants. All of a session's package instantiations (including state) can be lost if any of the session's instantiated packages (specification or body) are subsequently invalidated and recompiled.

Security Authorizations

Oracle notices when a DML object or system privilege is granted to or revoked from a user or PUBLIC and automatically invalidates all the owner's dependent

objects. Oracle invalidates the dependent objects to verify that an owner of a dependent object continues to have the necessary privileges for all referenced objects. Internally, Oracle notes that such objects do not have to be "recompiled"; only security authorizations need to be validated, not the structure of any objects. This optimization eliminates unnecessary recompilations and prevents the need to change a dependent object's timestamp.

> **Additional Information:** For information on forcing the recompilation of an invalid view or program unit, see the *Oracle8 Application Developer's Guide*. If you are using Trusted Oracle, also see your *Trusted Oracle* documentation.

Dependency Management and Nonexistent 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 *nonexistence* 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 the PUBLIC role.
- The JWARD schema does not contain a table or private synonym named EMP.
- The user JWARD creates a view in his schema with the following statement:

```
CREATE VIEW dept_salaries AS
   SELECT deptno, MIN(sal), AVG(sal), MAX(sal) FROM emp
   GROUP BY deptno
   ORDER BY deptno;
```

When JWARD creates the DEPT_SALARIES view, the reference to EMP is resolved by first looking for JWARD.EMP as a table, view, or private synonym, none of

which is found, and then as a public synonym named EMP, which is found. As a result, Oracle notes that JWARD.DEPT_SALARIES depends on the nonexistence of JWARD.EMP and on the existence of PUBLIC.EMP.

Now assume that JWARD decides to create a new view named EMP in his schema using the following statement:

```
CREATE VIEW emp AS
   SELECT empno, ename, mgr, deptno
   FROM company.emp;
```

Notice that JWARD.EMP does not have the same structure as COMPANY.EMP.

As it attempts to resolve references in object definitions, Oracle internally makes note of dependencies that the new dependent object has on "nonexistent" objects — schema objects that, if they existed, would change the interpretation of the object's definition. Such dependencies must be noted in case a nonexistent object is later created. If a nonexistent object is created, all dependent objects must be invalidated so that dependent objects can be recompiled and verified.

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 nonexistent objects checked during object resolution must be managed in case the nonexistent object is later created.

Shared SQL Dependency Management

In addition to managing dependencies among schema objects, Oracle also manages dependencies of each shared SQL area in the shared pool. If a table, view, synonym, or sequence is created, altered, or dropped, or a procedure or package specification is recompiled, all dependent shared SQL areas are invalidated. At a subsequent execution of the cursor that corresponds to an invalidated shared SQL area, Oracle reparses the SQL statement to regenerate the shared SQL area.

Local and Remote Dependency Management

Tracking dependencies and completing necessary recompilations are performed automatically by Oracle. In the simplest case, Oracle must manage dependencies among the objects in a single database (local dependency management). For exam-

ple, a statement in a procedure can reference a table in the same database. In more complex systems, Oracle must manage 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 dependencies using the database's internal "depends-on" table, which keeps track of each schema object's dependent objects. When a referenced object is modified, Oracle uses the depends-on table to identify dependent objects, which are then invalidated. For example, assume a stored procedure UPDATE_SAL references the table JWARD.EMP. If the definition of the table is altered in any way, the status of every object that references JWARD.EMP is changed to INVALID, including the stored procedure UPDATE_SAL. As a result, the procedure cannot be executed until it has been recompiled and is valid. Similarly, when a DML privilege is revoked from a user, every dependent object in the user's schema is invalidated. However, an object that is invalid because authorization was revoked can be revalidated by "reauthorization", in which case it does not require full recompilation.

Managing Remote Dependencies

Application-to-database and distributed database dependencies must also be managed. For example, an Oracle Forms application might contain a trigger that references a table, or a local stored procedure might call a remote procedure in a distributed database system. The database system must account for dependencies among such objects. Oracle uses different mechanisms to manage remote dependencies, depending on the objects involved.

Dependencies Among Local and Remote Database Procedures

Dependencies among stored procedures (including functions, packages, and triggers) in a distributed database system are managed using *timestamp checking* or *signature checking*.

The dynamic initialization parameter REMOTE_DEPENDENCIES_MODE determines whether timestamps or signatures govern remote dependencies.

> **Additional Information:** See *Oracle8 Application Developer's Guide* for details about managing remote dependencies with timestamps or signatures.

Timestamp Checking In the timestamp checking dependency model, whenever a procedure is compiled or recompiled its *timestamp* (the time it is created, altered, or replaced) is recorded in the data dictionary. Additionally, the compiled version of the procedure contains information about each remote procedure that it references, including the remote procedure's schema, package name, procedure name, and timestamp.

When a dependent procedure is used, Oracle compares the remote timestamps recorded at compile time with the current timestamps of the remotely referenced procedures. Depending on the result of this comparison, two situations can occur:

- The local and remote procedures execute without compilation if the timestamps match.
- The local procedure is invalidated if any timestamps of remotely referenced procedures do not match, and an error is returned to the calling environment. Furthermore, all other local procedures that depend on the remote procedure with the new timestamp are also invalidated. For example, assume several local procedures call a remote procedure, and the remote procedure is recompiled. When one of the local procedures is executed and notices the different timestamp of the remote procedure, every local procedure that depends on the remote procedure is invalidated.

Actual timestamp comparison occurs when a statement in the body of a local procedure executes a remote procedure; only at this moment are the timestamps compared via the distributed database's communications link. Therefore, all statements in a local procedure that precede an invalid procedure call might execute successfully. Statements subsequent to an invalid procedure call do not execute at all (compilation is required). However, any DML statements executed before the invalid procedure call are rolled back.

Signature Checking Oracle provides the additional capability of remote dependencies using *signatures*. The signature capability affects only remote dependencies. Local (same server) dependencies are not affected, as recompilation is always possible in this environment.

The signature of a procedure contains information about the

- name of the package, procedure, or function
- base types of the parameters
- modes of the parameters (IN, OUT, and IN OUT)

Note: Only the types and modes of parameters are significant. The name of the parameter does not affect the signature.

If the signature dependency model is in effect, a dependency on a remote program unit (package, stored procedure, stored function, or trigger) causes an invalidation of the dependent unit if the dependent unit contains a call to a procedure in the parent unit, and the signature of this procedure has been changed in an incompatible manner.

Dependencies Among Other Remote Schema Objects

Oracle does not manage dependencies among remote schema objects other than local-procedure-to-remote-procedure dependencies.

For example, assume that a local view is created and defined by a query that references a remote table. Also assume that a local procedure includes a SQL statement that references the same remote table. Later, the definition of the table is altered.

As a result, the local view and procedure are never invalidated, even if the view or procedure is used after the table is altered, and even if the view or procedure now returns errors when used (in this case, the view or procedure must be altered manually so errors are not returned). In such cases, lack of dependency management is preferable to unnecessary recompilations of dependent objects.

Dependencies of Applications

Code in database applications can reference objects in the connected database. For example, OCI, Precompiler, and SQL*Module applications can submit anonymous PL/SQL blocks; triggers in Oracle Forms applications can reference a schema object.

Such applications are dependent on the schema objects they reference. Dependency management techniques vary, depending on the development environment. Refer to the appropriate manuals for your application development tools and your operating system for more information about managing the remote dependencies within database applications.

20

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?
  - Execution Plans
  - Execution Order
- Cost-Based and Rule-Based Optimization
- Overview of Optimizer Operations
  - Evaluation of Expressions and Conditions
  - Transforming and Optimizing Statements
  - Choosing an Optimization Approach and Goal
  - Choosing Access Paths
  - Optimizing Join Statements
  - Optimizing Anti-Joins and Semi-Joins
  - Optimizing "Star" Queries

Additional Information: For more information on the Oracle optimizer, see *Oracle8 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 (DML) statement: SELECT, INSERT, UPDATE, or DELETE. Many different ways to execute a SQL statement often exist, for example, by varying the order in which tables or indexes are accessed. The procedure Oracle uses to execute a statement can greatly affect how quickly the statement executes.

A part of Oracle called the *optimizer* chooses what it believes to be the most efficient way. The optimizer evaluates a number of factors to select among alternative access paths. Sometimes the application designer, who has more information about a particular application's data than is available to the optimizer, can choose a more effective way to execute a SQL statement. The application designer can use hints in SQL statements to specify how the statement should be executed.

Note: The optimizer may not make the same decisions from one version of Oracle to the next. In more recent versions, the optimizer may make different decisions based on better, more sophisticated information available to it.

Additional Information: See *Oracle8 Tuning* for information about using hints in SQL statements.

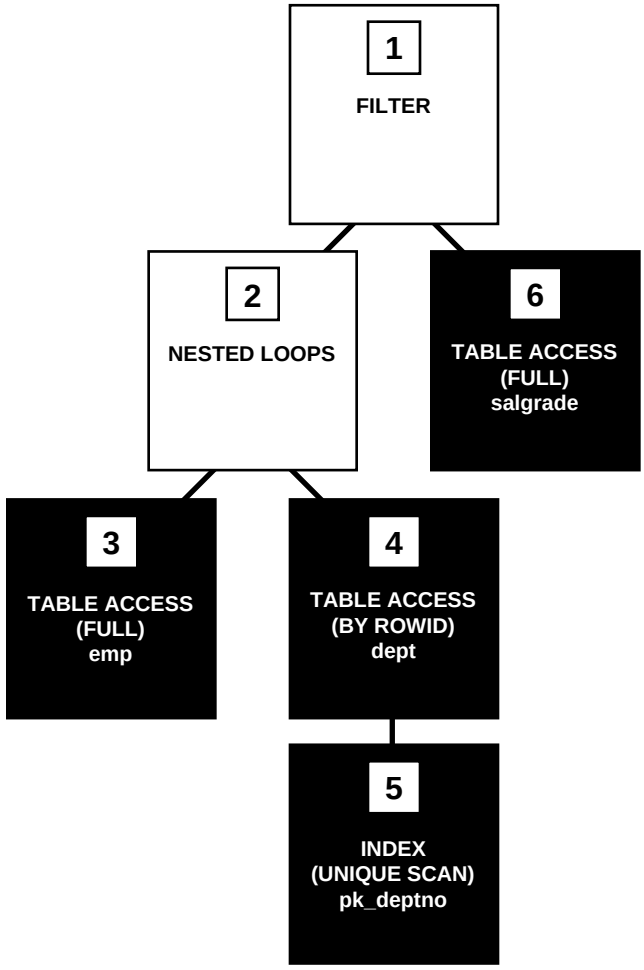
Execution Plans

To execute a DML statement, Oracle may have to perform many steps. Each of these steps either retrieves rows of data physically from the database or prepares them in some way for the user issuing the statement. The combination of the steps Oracle uses to execute a statement is called an *execution plan*.

Figure 20–1 shows a graphical representation of the execution plan for the following SQL statement, which selects the name, job, salary, and department name for all employees whose salaries do not fall into a recommended salary range:

```
SELECT ename, job, sal, dname
    FROM emp, dept
  WHERE emp.deptno = dept.deptno
    AND NOT EXISTS
      (SELECT *
           FROM salgrade
         WHERE emp.sal BETWEEN losal AND hisal);
```

Figure 20–1 An Execution Plan



Steps of Execution Plan

Each step of the execution plan returns a set of rows that either are used by the next step or, in the last step, are returned to the user or application issuing the SQL statement. A set of rows returned by a step is called a *row source*.

Figure 20–1 is a hierarchical diagram showing the flow of row sources from one step to another. The numbering of the steps reflects the order in which they are dis-

played in response to the EXPLAIN PLAN command (described in the next section). This generally is **not** the order in which the steps are executed (see "Execution Order" on page 20-5).

Each step of the execution plan either retrieves rows from the database or accepts rows from one or more row sources as input:

- Steps indicated by the shaded boxes physically retrieve data from an object in the database. Such steps are called *access paths*:
  - Steps 3 and 6 read all the rows of the EMP and SALGRADE tables, respectively.
  - Step 5 looks up in the PK_DEPTNO index each DEPTNO value returned by Step 3. There it finds the ROWIDs of the associated rows in the DEPT table.
  - Step 4 retrieves from the DEPT table the rows whose ROWIDs were returned by Step 5.
- Steps indicated by the clear boxes operate on row sources:
  - Step 2 performs a nested loops operation, accepting row sources from Steps 3 and 4, joining each row from Step 3 source to its corresponding row in Step 4, and returning the resulting rows to Step 1.
  - Step 1 performs a filter operation. It accepts row sources from Steps 2 and 6, eliminates rows from Step 2 that have a corresponding row in Step 6, and returns the remaining rows from Step 2 to the user or application issuing the statement.

Access paths are discussed further in the section "Choosing Access Paths" on page 20-42. Methods by which Oracle joins row sources are discussed in "Join Operations" on page 20-63.

The EXPLAIN PLAN Command

You can examine the execution plan chosen by the optimizer for a SQL statement by using the EXPLAIN PLAN command, which causes the optimizer to choose the execution plan and then inserts data describing the plan into a database table.

For example, the following output table is such a description for the statement examined in the previous section:

```
ID    OPERATION           OPTIONS        OBJECT_NAME
-----------------------------------------------------------
0     SELECT STATEMENT
1       FILTER
2         NESTED LOOPS
3           TABLE ACCESS  FULL           EMP
4           TABLE ACCESS  BY ROWID       DEPT
5             INDEX       UNIQUE SCAN    PK_DEPTNO
6           TABLE ACCESS  FULL           SALGRADE
```

Each box in Figure 20–1 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 20–1.

You can obtain such a listing by using the EXPLAIN PLAN command and then querying the output table.

> **Additional Information:** See *Oracle8 Tuning* for information on how to use EXPLAIN PLAN and produce and interpret its output.

Execution Order

The steps of the execution plan are not performed in the order in which they are numbered. Rather, Oracle first performs the steps that appear as leaf nodes in the tree-structured graphical representation of the execution plan (Steps 3, 5, and 6 in Figure 20–1). 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 20–1, 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, joining the single row from Step 3 with 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 not returned from Step 6, Oracle returns the row from Step 2 to the user issuing the SQL statement.

Note that Oracle performs Steps 5, 4, 2, 6, and 1 once for each row returned by Step 3. If a parent step requires only a single row from its child step before it can be executed, Oracle performs the parent step (and possibly the rest of the execution plan) as soon as a single row has been returned from the child step. If the parent of that parent step also can be activated by the return of a single row, then it is executed as well.

Thus the execution can cascade up the tree, 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.

If a parent step requires all rows from its child step before it can be executed, Oracle cannot perform the parent step until all rows have been returned from the child step. Such parent steps include sorts, sort-merge joins, group functions, and aggregates.

Cost-Based and Rule-Based Optimization

To choose an execution plan for a SQL statement, the optimizer uses one of two approaches: cost-based or rule-based.

The Cost-Based Approach

Using the cost-based approach, the optimizer determines which execution plan is most efficient by considering available access paths and factoring in information based on statistics in the data dictionary for the schema objects (tables, clusters, or indexes) accessed by the statement. The cost-based approach also considers hints, or optimization suggestions placed in a Comment in the statement.

Conceptually, the cost-based approach consists of these steps:

1. The optimizer generates a set of potential execution plans for the 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 resource use 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, that are required to execute the statement using the plan.

Serial execution plans with greater costs take more time to execute than those with smaller costs. When using a parallel execution plan, however, resource use is not directly related to elapsed time.

3. The optimizer compares the costs of the execution plans and chooses the one with the smallest cost.

Goal of the Cost-Based Approach

By default, the goal of the cost-based approach is the best *throughput*, or minimal resource use necessary to process all rows accessed by the statement.

Oracle can also optimize a statement with the goal of best *response time*, or minimal resource use necessary to process the first row accessed by a SQL statement. For information on how the optimizer chooses an optimization approach and goal, see "Choosing an Optimization Approach and Goal" on page 20-40.

Note: For parallel execution, the optimizer can choose to minimize elapsed time at the expense of resource consumption. Use the initialization parameter OPTIMIZER_PERCENT_PARALLEL to specify how much the optimizer attempts to parallelize.

Additional Information: See *Oracle8 Tuning* for information about using the OPTIMIZER_PERCENT_PARALLEL parameter.

Statistics 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, indexes, and partitions. You can generate these statistics using the ANALYZE command. The optimizer uses these statistics to estimate how much I/O, CPU time, and memory are required to execute a SQL statement using a particular execution plan.

You can view the statistics with 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

- USER_TAB_PARTITIONS, ALL_TAB_PARTITIONS, and DBA_TAB_PARTITIONS
- USER_IND_PARTITIONS, ALL_IND_PARTITIONS, and DBA_IND_PARTITIONS
- USER_PART_COL_STATISTICS, ALL_PART_COL_STATISTICS, and DBA_PART_COL_STATISTICS

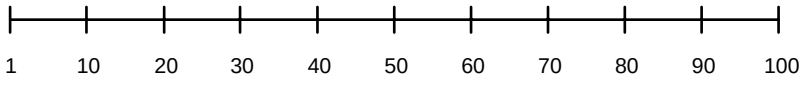
> **Additional Information:** For information on these statistics, see the *Oracle8 Reference*.

Histograms

Oracle's cost-based optimizer 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 nonuniform data distributions. You generate histograms by using the ANALYZE command.

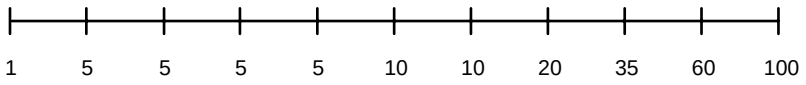
One of the fundamental capabilities of any cost-based optimizer is determining the selectivity of predicates that appear in queries. Selectivity estimates are used to decide when to use an index and the order in which to join tables. Most attribute domains (a table's columns) are *not* uniformly distributed. The Oracle cost-based optimizer uses height-balanced histograms on specified attributes to describe the distributions of nonuniform domains.

Histogram Examples Consider a column C with values between 1 and 100 and a histogram with 10 buckets. If the data in C is uniformly distributed, this histogram would look like this, where the numbers are the endpoint values.



The number of rows in each bucket is one tenth the total number of rows in the table. Four-tenths of the rows have values between 60 and 100 in this example of uniform distribution.

If the data is not uniformly distributed, the histogram might look like this:



In this case, most of the rows have the value 5 for the column. In this example, only 1/10 of the rows have values between 60 and 100.

Height-Balanced Histograms Oracle uses *height-balanced* histograms (as opposed to *width-balanced*).

- Width-balanced histograms divide the data into a fixed number of equal-width ranges and then count the number of values falling into each range.
- Height-balanced histograms place the same number of values into each range so that the endpoints of the range are determined by how many values are in that range.

For example, suppose that the values in a single column of a 1000-row table range between 1 and 100, and suppose that you want a 10-bucket histogram (ranges in a histogram are called *buckets*). In a width-balanced histogram, the buckets would be of equal width (1-10, 11-20, 21-30, and so on) and each bucket would count the number of rows that fall into that bucket's range. In a height-balanced histogram, each bucket has the same height (in this case 100 rows) and the endpoints for each bucket are determined by the density of the distinct values in the column.

Advantages of Height-Balanced Histograms The advantage of the height-balanced approach is clear when the data is highly skewed. Suppose that 800 rows of a 1000-row table have the value 5, and the remaining 200 rows are evenly distributed between 1 and 100. A width-balanced histogram would have 820 rows in the bucket labeled 1-10 and approximately 20 rows in each of the other buckets. The height-based histogram would have one bucket labeled 1-5, seven buckets labeled 5-5, one bucket labeled 5-50, and one bucket labeled 50-100.

If you want to know how many rows in the table contain the value 5, it is apparent from the height-balanced histogram that approximately 80% of the rows contain this value. However, the width-balanced histogram does not provied a mechanism for differentiating between the value 5 and the value 6. You would compute only 8% of the rows contain the value 5 in a width-balanced histogram. Therefore height-

based histograms are more appropriate for determining the selectivity of column values.

When to Use Histograms For many users, it is appropriate to use the FOR ALL INDEXED COLUMNS option of the ANALYZE command to create histograms because indexed columns are typically the columns most often used in WHERE clauses.

You can view histograms with the following views:

- USER_HISTOGRAMS, ALL_HISTOGRAMS, and DBA_HISTOGRAMS
- USER_PART_HISTOGRAMS, ALL_PART_HISTOGRAMS, and DBA_PART_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.

> **Additional Information:** See *Oracle8 Tuning* for more information about histograms.

When to Use the Cost-Based Approach In general, you should use the cost-based approach for all new applications; the rule-based approach is provided for applications that were written before cost-based optimization was available. Cost-based optimization can be used for both relational data and object types.

The following features can only use cost-based optimization:

- partitioned tables
- partition views
- index-organized tables

- reverse key indexes
- bitmap indexes
- parallel query and parallel DML
- star transformation
- star join

> **Additional Information:** See *Oracle8 Tuning* for more information on when to use the cost-based approach.

The Rule-Based Approach

Using the rule-based approach, the optimizer chooses an execution plan based on the access paths available and the ranks of these access paths (shown in Table 20–1 "Access Paths" on page 20-46). You can use rule-based optimization to access both relational data and object types.

Oracle's ranking of the access paths is heuristic. If there is more than one way to execute a SQL statement, the rule-based approach always uses the operation with the lower rank. Usually, operations of lower rank execute faster than those associated with constructs of higher rank.

For more information, see "Choosing Among Access Paths with the Rule-Based Approach" on page 20-62.

Overview of Optimizer Operations

This section summarizes the operations performed by the Oracle optimizer and describes the types of SQL statements that can be optimized.

Optimizer Operations

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. (See "Evaluation of Expressions and Conditions" on page 20-14.)
statement transformation	For a complex statement involving, for example, correlated subqueries, the optimizer may transform the original statement into an equivalent join statement. (See "Transforming and Optimizing Statements" on page 20-19.)
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. (See "Optimizing Statements That Access Views" on page 20-24.)
choice of optimization approaches	The optimizer chooses either a cost-based or rule-based approach to optimization and determines the goal of optimization. (See "Choosing an Optimization Approach and Goal" on page 20-40.)
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. (See "Choosing Access Paths" on page 20-42.)
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, and so on. (See "Optimizing Join Statements" on page 20-63.)
choice of join operations	For any join statement, the optimizer chooses an operation to use to perform the join. (See "Optimizing Join Statements" on page 20-63.)

Types of SQL Statements

Oracle optimizes these different types of SQL statements:

simple statement	An INSERT, UPDATE, DELETE, or SELECT statement that involves only a single table.
simple query	Another name for a SELECT statement.
join	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.
equijoin	A join condition containing an equality operator.
nonequijoin	A join condition containing something other than an equality operator.
outer join	A join condition using the outer join operator (+) with one or more 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 product	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 statement	An INSERT, UPDATE, DELETE, or SELECT statement that contains a subquery, which is a form of the SELECT statement 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 query	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*.
statement accessing views	Simple, join, complex, or compound statement that accesses one or more views as well as tables.
distributed statement	A statement that accesses data on a remote database.

Evaluation of Expressions and Conditions

The optimizer fully evaluates expressions whenever possible and translates certain syntactic constructs into equivalent constructs. The reason for this is either that Oracle can more quickly evaluate the resulting expression than the original expression, or that the original expression is merely a syntactic equivalent of the resulting expression. Different SQL constructs can sometimes operate identically (for example, = ANY (*subquery*) and IN (*subquery*)); Oracle maps these to a single construct.

Constants

Computation of constants is performed only once, when the statement is optimized, rather than each time the statement is executed.

Consider these conditions that test for monthly salaries greater than 2000:

```
sal > 24000/12

sal > 2000

sal*12 > 24000
```

If a SQL statement contains the first condition, the optimizer simplifies it into the second condition.

Note that the optimizer does not simplify expressions across comparison operators: in the examples above, the optimizer does not simplify the third expression into the second. For this reason, application developers should write conditions that compare columns with constants whenever possible, rather than conditions with expressions involving columns.

LIKE Operator

The optimizer simplifies conditions that use the LIKE comparison operator to compare an expression with no wildcard characters into an equivalent condition that uses an equality operator instead. For example, the optimizer simplifies the first condition below into the second:

```
ename LIKE 'SMITH'

ename = 'SMITH'
```

The optimizer can simplify these expressions only when the comparison involves variable-length datatypes. For example, if ENAME was of type CHAR(10), the optimizer cannot transform the LIKE operation into an equality operation due to the equality operator following blank-padded semantics and LIKE not following blank-padded semantics.

IN Operator

The optimizer expands a condition that uses the IN comparison operator to an equivalent condition that uses equality comparison operators and OR logical operators. For example, the optimizer expands the first condition below into the second:

```
ename IN ('SMITH', 'KING', 'JONES')

ename = 'SMITH' OR ename = 'KING' OR ename = 'JONES'
```

See “Example 2: IN Subquery” on page 20-26 for more information.

ANY or SOME Operator

The optimizer expands a condition that uses the ANY or SOME comparison operator followed by a parenthesized list of values into an equivalent condition that uses equality comparison operators and OR logical operators. For example, the optimizer expands the first condition below into the second:

```
sal > ANY (:first_sal, :second_sal)

sal > :first_sal OR sal > :second_sal
```

The optimizer transforms a condition that uses the ANY or SOME operator followed by a subquery into a condition containing the EXISTS operator and a correlated subquery. For example, the optimizer transforms the first condition below into the second:

```
x > ANY (SELECT sal
    FROM emp
    WHERE job = 'ANALYST')

EXISTS (SELECT sal
    FROM emp
    WHERE job = 'ANALYST'
      AND x > sal)
```

ALL Operator

The optimizer expands a condition that uses the ALL comparison operator followed by a parenthesized list of values into an equivalent condition that uses equality comparison operators and AND logical operators. For example, the optimizer expands the first condition below into the second:

```
sal > ALL (:first_sal, :second_sal)

sal > :first_sal AND sal > :second_sal
```

The optimizer transforms a condition that uses the ALL comparison operator followed by a subquery into an equivalent condition that uses the ANY comparison operator and a complementary comparison operator. For example, the optimizer transforms the first condition below into the second:

```
x > ALL (SELECT sal
    FROM emp
    WHERE deptno = 10)

NOT (x <= ANY (SELECT sal
      FROM emp
      WHERE deptno = 10) )
```

The optimizer then transforms the second query into the following query using the rule for transforming conditions with the ANY comparison operator followed by a correlated subquery:

```
NOT EXISTS (SELECT sal
      FROM emp
      WHERE deptno = 10
        AND x <= sal)
```

BETWEEN Operator

The optimizer always replaces a condition that uses the BETWEEN comparison operator with an equivalent condition that uses the >= and <= comparison operators. For example, the optimizer replaces the first condition below with the second:

```
sal BETWEEN 2000 AND 3000

sal >= 2000 AND sal <= 3000
```

NOT Operator

The optimizer simplifies a condition to eliminate the NOT logical operator. The simplification involves removing the NOT logical operator and replacing a comparison operator with its opposite comparison operator. For example, the optimizer simplifies the first condition below into the second one:

```
NOT deptno = (SELECT deptno FROM emp WHERE ename = 'TAYLOR')

deptno <> (SELECT deptno FROM emp WHERE ename = 'TAYLOR')
```

Often a condition containing the NOT logical operator can be written many different ways. The optimizer attempts to transform such a condition so that the subconditions negated by NOTs are as simple as possible, even if the resulting condition contains more NOTs. For example, the optimizer simplifies the first condition below into the second and dhen into the third.

```
NOT (sal < 1000 OR comm IS NULL)

NOT sal < 1000 AND comm IS NOT NULL

sal >= 1000 AND comm IS NOT NULL
```

Transitivity

If two conditions in the WHERE clause involve a common column, the optimizer can sometimes infer a third condition using the transitivity principle. The optimizer can then use the inferred condition to optimize the statement. The inferred condition could potentially make available an index access path that was not made available by the original conditions.

Note: Transitivity is used only by the cost-based approach.

Imagine a WHERE clause containing two conditions of these forms:

```
WHERE column1 comp_oper constant
  AND column1 = column2
```

In this case, the optimizer infers the condition:

```
column2 comp_oper constant
```

where:

comp_oper	is any of the comparison operators =, !=, ^=, <, <>, >, <=, or >=.
constant	is any constant expression involving operators, SQL functions, literals, bind variables, and correlation variables.

Example: Consider this query in which the WHERE clause contains two conditions, each or which uses the EMP.DEPTNO column:

```
SELECT *
  FROM emp, dept
  WHERE emp.deptno = 20
    AND emp.deptno = dept.deptno;
```

Using transitivity, the optimizer infers this condition:

```
dept.deptno = 20
```

If an index exists on the DEPT.DEPTNO column, this condition makes available access paths using that index.

Note: The optimizer only infers conditions that relate columns to constant expressions, rather than columns to other columns. Imagine a WHERE clause containing two conditions of these forms:

```
WHERE column1 comp_oper column3
  AND column1 = column2
```

In this case, the optimizer does not infer this condition:

```
column2 comp_oper column3
```

Transforming and Optimizing Statements

SQL is a very flexible query language; there are often many statements you could formulate to achieve the same goal. Sometimes the optimizer transforms one such statement into another that achieves the same goal if the second statement can be executed more efficiently.

This section discusses the following topics:

- Transforming ORs into Compound Queries
- Transforming Complex Statements into Join Statements
- Optimizing Statements That Access Views
- Optimizing Compound Queries
- Optimizing Distributed Statements

For additional information about optimizing statements, see "Optimizing Join Statements" on page 20-63 and "Optimizing "Star" Queries" on page 20-75.

Transforming ORs into Compound Queries

If a query contains a WHERE clause with multiple conditions combined with OR operators, the optimizer transforms it into an equivalent compound query that uses the UNION ALL set operator if this makes it execute more efficiently:

- If each condition individually makes an index access path available, the optimizer can make the transformation. The optimizer then chooses an execution plan for the resulting statement that accesses the table multiple times using the different indexes and then puts the results together.
- If any condition requires a full table scan because it does not make an index available, the optimizer does not transform the statement. The optimizer chooses a full table scan to execute the statement, and Oracle tests each row in the table to determine whether it satisfies any of the conditions.
- For statements that use the cost-based approach, the optimizer may use statistics to determine whether to make the transformation by estimating and then comparing the costs of executing the original statement versus the resulting statement.
- The cost-based optimizer does not use the OR transformation for in lists or ORs on the same column; instead, it uses the inlist iterator operator.

Additional Information: For more information, see *Oracle8 Tuning*.

For information on access paths and how indexes make them available, see Table 20–1 “Access Paths” on page 20-46 and the sections that follow it.

Example: Consider this query with a WHERE clause that contains two conditions combined with an OR operator:

```
SELECT *
  FROM emp
  WHERE job = 'CLERK'
    OR deptno = 10;
```

If there are indexes on both the JOB and DEPTNO columns, the optimizer may transform this query into the equivalent query below:

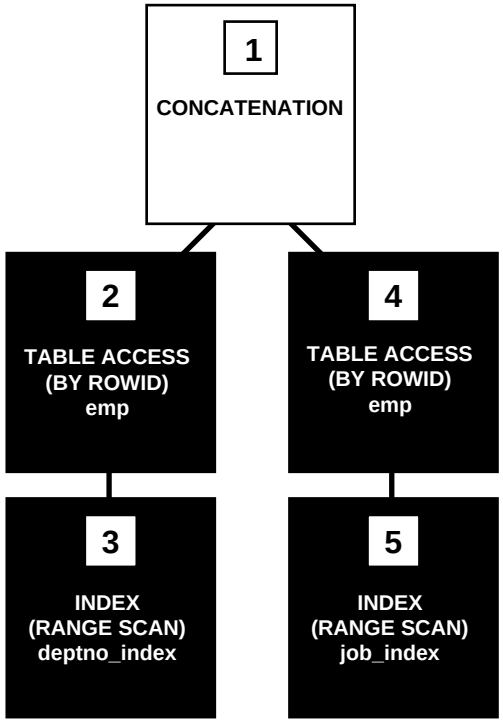
```
SELECT *
  FROM emp
  WHERE job = 'CLERK'
UNION ALL
SELECT *
  FROM emp
  WHERE deptno = 10
    AND job <> 'CLERK';
```

If you are using the cost-based approach, the optimizer compares the cost of executing the original query using a full table scan with that of executing the resulting query when deciding whether to make the transformation.

If you are using the rule-based approach, the optimizer makes this UNION ALL transformation because each component query of the resulting compound query can be executed using an index. The rule-based approach assumes that executing the compound query using two index scans is faster than executing the original query using a full table scan.

The execution plan for the transformed statement might look like the illustration in Figure 20–2:

Figure 20–2 Execution Plan for a Transformed Query Containing OR



To execute the transformed query, Oracle performs the following steps:

- Steps 3 and 5 scan the indexes on the JOB and DEPTNO columns using the conditions of the component queries. These steps obtain ROWIDs of the rows that satisfy the component queries.
- Steps 2 and 4 use the ROWIDs from Steps 3 and 5 to locate the rows that satisfy each component query.
- Step 1 puts together the row sources returned by Steps 2 and 4.

If either of the JOB or DEPTNO columns is not indexed, the optimizer does not even consider the transformation, because the resulting compound query would require a full table scan to execute one of its component queries. Executing the compound query with a full table scan in addition to an index scan could not possibly be faster than executing the original query with a full table scan.

Example: Consider this query and assume that there is an index on the ENAME column only:

```
SELECT *
  FROM emp
  WHERE ename = 'SMITH'
    OR sal > comm;
```

Transforming the query above would result in the compound query below:

```
SELECT *
  FROM emp
  WHERE ename = 'SMITH'
UNION ALL
SELECT *
  FROM emp
  WHERE sal > comm;
```

Since the condition in the WHERE clause of the second component query (SAL > COMM) does not make an index available, the compound query requires a full table scan. For this reason, the optimizer does not make the transformation and it chooses a full table scan to execute the original statement.

Transforming Complex Statements into Join Statements

To optimize a complex statement, the optimizer chooses one of these alternatives:

- Transform the complex statement into an equivalent join statement and then optimize the join statement.
- Optimize the complex statement as is.

The optimizer transforms a complex statement into a join statement whenever the resulting join statement is guaranteed to return exactly the same rows as the complex statement. This transformation allows Oracle to execute the statement by taking advantage of join optimization techniques described in "Optimizing Join Statements" on page 20-63.

Consider this complex statement that selects all rows from the ACCOUNTS table whose owners appear in the CUSTOMERS table:

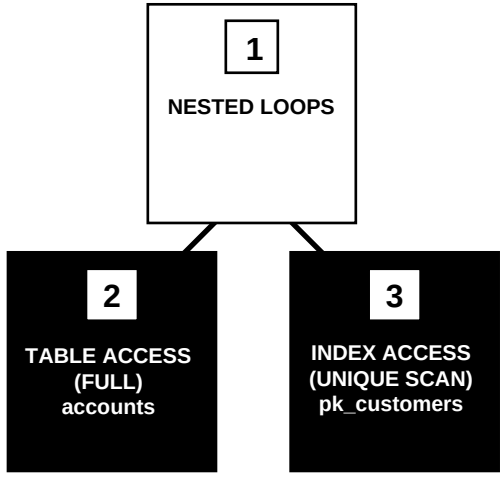
```
SELECT *
  FROM accounts
  WHERE custno IN
    (SELECT custno FROM customers);
```

If the CUSTNO column of the CUSTOMERS table is a primary key or has a UNIQUE constraint, the optimizer can transform the complex query into this join statement that is guaranteed to return the same data:

```
SELECT accounts.*
  FROM accounts, customers
  WHERE accounts.custno = customers.custno;
```

The execution plan for this statement might look like Figure 20–3.

Figure 20–3 Execution Plan for a Nested Loops Join



To execute this statement, Oracle performs a nested-loops join operation. For information on nested loops joins, see “Join Operations” on page 20-63.

If the optimizer cannot transform a complex statement into a join statement, the optimizer chooses execution plans for the parent statement and the subquery as though they were separate statements. Oracle then executes the subquery and uses the rows it returns to execute the parent query.

Consider this complex statement that returns all rows from the ACCOUNTS table that have balances greater than the average account balance:

```
SELECT *
  FROM accounts
  WHERE accounts.balance >
    (SELECT AVG(balance) FROM accounts);
```

No join statement can perform the function of this statement, so the optimizer does not transform the statement. Note that complex queries whose subqueries contain group functions such as AVG cannot be transformed into join statements.

Optimizing Statements That Access Views

To optimize a statement that accesses a view, the optimizer chooses one of these alternatives:

- Transform the statement into an equivalent statement that accesses the view's base tables, then optimize the resulting statement. The optimizer can use one of these techniques to transform the statement:
  - Merge the view's query into the referencing query block in the accessing statement.
  - Push the predicate of the referencing query block inside the view (for an unmergeable view).
- Issue the view's query, collecting all the returned rows, and then access this set of rows with the original statement as though it were a table. (See "Accessing the View's Rows with the Original Statement" on page 20-34.)

Merging the View's Query into the Statement

To merge the view's query into a referencing query block in the accessing statement, the optimizer replaces the name of the view with the names of its base tables in the query block and adds the condition of the view's query's WHERE clause to the accessing query block's WHERE clause.

This optimization applies to *select-project-join* views, which are views that contain only selections, projections, and joins — that is, views that do not contain set operators, group functions, DISTINCT, GROUP BY, CONNECT BY, and so on (as described in "Mergeable and Unmergeable Views" on page 20-25).

Example: Consider this view of all employees who work in department 10:

```
CREATE VIEW emp_10
  AS SELECT empno, ename, job, mgr, hiredate, sal, comm, deptno
    FROM emp
    WHERE deptno = 10;
```

Consider this query that accesses the view. The query selects the IDs greater than 7800 of employees who work in department 10:

```
SELECT empno
  FROM emp_10
  WHERE empno > 7800;
```

The optimizer transforms the query into the following query that accesses the view's base table:

```
SELECT empno
  FROM emp
  WHERE deptno = 10
    AND empno > 7800;
```

If there are indexes on the DEPTNO or EMPNO columns, the resulting WHERE clause makes them available.

Mergeable and Unmergeable Views The optimizer can merge a view into a referencing query block when the view has one or more base tables, provided the view does not contain:

- set operators (UNION, UNION ALL, INTERSECT, MINUS)
- a CONNECT BY clause
- a ROWNUM pseudocolumn
- group functions (AVG, COUNT, MAX, MIN, SUM) in the select list.

When a view contains one of the following structures, it can be merged into a referencing query block only if *complex view merging* is enabled (as described below):

- a GROUP BY clause
- a DISTINCT operator in the select list

View merging is not possible for a view that has multiple base tables if it is on the right side of an outer join. If a view on the right side of an outer join has only one base table, however, the optimizer can use complex view merging even if an expression in the view can return a non-null value for a NULL. See "Views in Outer Joins" on page 20-72 for more information.

Complex View Merging If a view's query contains a GROUP BY clause or DISTINCT operator in the select list, then the optimizer can merge the view's query into the accessing statement *only if* complex view merging is enabled. Complex merging can also be used to merge an IN subquery into the accessing statement, if the subquery is uncorrelated (see "Example 2: IN Subquery" on page 20-26).

Complex merging is not cost-based — it must be enabled with the initialization parameter COMPLEX_VIEW_MERGING or the MERGE hint, that is, either the COMPLEX_VIEW_MERGING parameter must be set to TRUE or the accessing query block must include the MERGE hint. Without this hint or parameter setting, the optimizer uses another approach (see "Pushing the Predicate into the View" on page 20-27).

> **Additional Information:** See *Oracle8 Tuning* for details about the MERGE and NO_MERGE hints.

Example 1: View with a GROUP BY Clause Consider the view AVG_SALARY_VIEW, which contains the average salaries for each department:

```
CREATE VIEW avg_salary_view AS
  SELECT deptno, AVG(sal) AS avg_sal_dept,
    FROM emp
    GROUP BY deptno;
```

If complex view merging is enabled then the optimizer can transform this query, which finds the average salaries of departments in London:

```
SELECT dept.deptloc, avg_sal_dept
  FROM dept, avg_salary_view
  WHERE dept.deptno = avg_salary_view.deptno
    AND dept.deptloc = 'London';
```

into this query:

```
SELECT dept.deptloc, AVG(sal)
  FROM dept, emp
  WHERE dept.deptno = emp.deptno
    AND dept.deptloc = 'London'
  GROUP BY dept.rowid, dept.deptloc;
```

The transformed query accesses the view's base table, selecting only the rows of employees who work in London and grouping them by department.

Example 2: IN Subquery Complex merging can be used for an IN clause with a non-correlated subquery, as well as for views. Consider the view MIN_SALARY_VIEW, which contains the minimum salaries for each department:

```
SELECT deptno, MIN(sal)
  FROM emp
  GROUP BY deptno;
```

If complex merging is enabled then the optimizer can transform this query, which finds all employees who earn the minimum salary for their department in London:

```
SELECT emp.ename, emp.sal
  FROM emp, dept
  WHERE (emp.deptno, emp.sal) IN min_salary_view
    AND emp.deptno = dept.deptno
    AND dept.deptloc = 'London';
```

into this query (where E1 and E2 represent the EMP table as it is referenced in the accessing query block and the view's query block, respectively):

```
SELECT e1.ename, e1.sal
  FROM emp e1, dept, emp e2
  WHERE e1.deptno = dept.deptno
    AND dept.deptloc = 'London'
    AND e1.deptno = e2.deptno
  GROUP BY e1.rowid, dept.rowid, e1.ename, e1.sal
  HAVING e1.sal = MIN(e2.sal);
```

Pushing the Predicate into the View

The optimizer can transform a query block that accesses an unmergeable view by pushing the query block's predicates inside the view's query.

Example 1: Consider the TWO_EMP_TABLES view, which is the union of two employee tables. The view is defined with a compound query that uses the UNION set operator:

```
CREATE VIEW two_emp_tables
  (empno, ename, job, mgr, hiredate, sal, comm, deptno) AS
  SELECT empno, ename, job, mgr, hiredate, sal, comm, deptno
    FROM emp1
  UNION
  SELECT empno, ename, job, mgr, hiredate, sal, comm, deptno
    FROM emp2;
```

Consider this query that accesses the view. The query selects the IDs and names of all employees in either table who work in department 20:

```
SELECT empno, ename
  FROM two_emp_tables
  WHERE deptno = 20;
```

Because the view is defined as a compound query, the optimizer cannot merge the view's query into the accessing query block. Instead, the optimizer can transform the accessing statement by pushing its predicate, the WHERE clause condition (DEPTNO = 20), into the view's compound query.

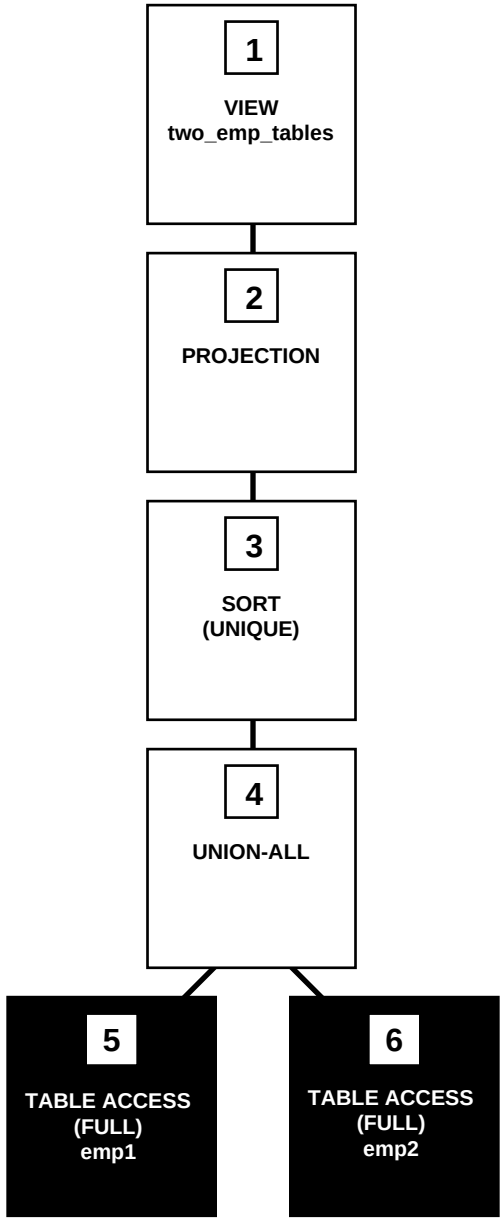
The resulting statement looks like this:

```
SELECT empno, ename
  FROM ( SELECT empno, ename, job, mgr, hiredate, sal, comm, deptno
           FROM emp1
           WHERE deptno = 20
         UNION
         SELECT empno, ename, job, mgr, hiredate, sal, comm, deptno
           FROM emp2
           WHERE deptno = 20 );
```

If there is an index on the DEPTNO column, the resulting WHERE clauses make it available.

Figure 20–4, "Accessing a View Defined with the UNION Set Operator", shows the execution plan of the resulting statement.

Figure 20–4 Accessing a View Defined with the UNION Set Operator



To execute this statement, Oracle performs these steps:

- Steps 5 and 6 perform full scans of the EMP1 and EMP2 tables.
- Step 4 performs a UNION-ALL operation returning all rows returned by either Step 5 or Step 6, including all copies of duplicates.
- Step 3 sorts the result of Step 4, eliminating duplicate rows.
- Step 2 extracts the desired columns from the result of Step 3.
- Step 1 indicates that the view's query was not merged into the accessing query.

Example 2: Consider the view EMP_GROUP_BY_DEPTNO, which contains the department number, average salary, minimum salary, and maximum salary of all departments that have employees:

```
CREATE VIEW emp_group_by_deptno
  AS SELECT deptno,
         AVG(sal) avg_sal,
         MIN(sal) min_sal,
         MAX(sal) max_sal
    FROM emp
    GROUP BY deptno;
```

Consider this query, which selects the average, minimum, and maximum salaries of department 10 from the EMP_GROUP_BY_DEPTNO view:

```
SELECT *
  FROM emp_group_by_deptno
  WHERE deptno = 10;
```

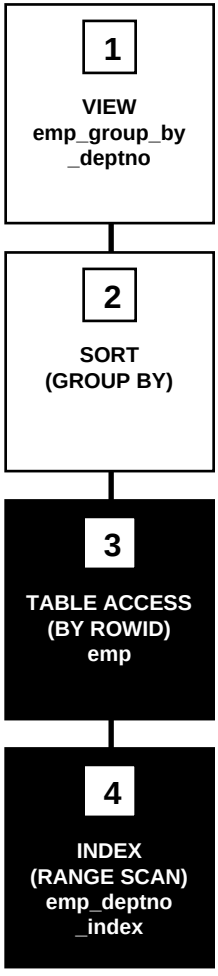
The optimizer transforms the statement by pushing its predicate (the WHERE clause condition) into the view's query. The resulting statement looks like this:

```
SELECT deptno,
    AVG(sal) avg_sal,
    MIN(sal) min_sal,
    MAX(sal) max_sal,
  FROM emp
  WHERE deptno = 10
  GROUP BY deptno;
```

If there is an index on the DEPTNO column, the resulting WHERE clause makes it available.

Figure 20–5, “Accessing a View Defined with a GROUP BY Clause”, shows the execution plan for the resulting statement. The execution plan uses an index on the DEPTNO column.

Figure 20–5 Accessing a View Defined with a GROUP BY Clause



To execute this statement, Oracle performs these operations:

- Step 4 performs a range scan on the index EMP_DEPTNO_INDEX (an index on the DEPTNO column of the EMP table) to retrieve the ROWIDs of all rows in the EMP table with a DEPTNO value of 10.
- Step 3 accesses the EMP table using the ROWIDs retrieved by Step 4.
- Step 2 sorts the rows returned by Step 3 to calculate the average, minimum, and maximum SAL values.
- Step 1 indicates that the view's query was not merged into the accessing query.

Applying a Group Function to the View The optimizer can transform a query that contains a group function (AVG, COUNT, MAX, MIN, SUM) by applying the function to the view's query.

Example: Consider a query that accesses the EMP_GROUP_BY_DEPTNO view defined in the previous example. This query derives the averages for the average department salary, the minimum department salary, and the maximum department salary from the employee table:

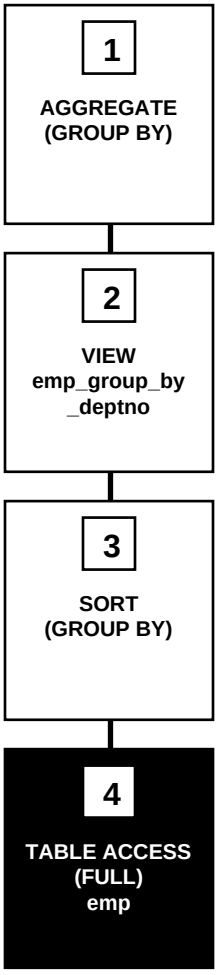
```
SELECT AVG(avg_sal), AVG(min_sal), AVG(max_sal)
    FROM emp_group_by_deptno;
```

The optimizer transforms this statement by applying the AVG group function to the select list of the view's query:

```
SELECT AVG(AVG(sal)), AVG(MIN(sal)), AVG(MAX(sal))
    FROM emp
    GROUP BY deptno;
```

Figure 20–6 shows the execution plan of the resulting statement.

Figure 20–6 Applying Group Functions to a View Defined with GROUP BY Clause



To execute this statement, Oracle performs these operations:

- Step 4 performs a full scan of the EMP table.
- Step 3 sorts the rows returned by Step 4 into groups based on their DEPTNO values and calculates the average, minimum, and maximum SAL value of each group.

- Step 2 indicates that the view's query was not merged into the accessing query.
- Step 1 calculates the averages of the values returned by Step 2.

Accessing the View's Rows with the Original Statement

The optimizer cannot transform all statements that access views into equivalent statements that access base table(s). For example, if a query accesses a ROWNUM pseudocolumn in a view, the view cannot be merged into the query and the query's predicate cannot be pushed into the view.

To execute a statement that cannot be transformed into one that accesses base tables, Oracle issues the view's query, collects the resulting set of rows, and then accesses this set of rows with the original statement as though it were a table.

Example: Consider the EMP_GROUP_BY_DEPTNO view defined in the previous section:

```
CREATE VIEW emp_group_by_deptno
  AS SELECT deptno,
        AVG(sal) avg_sal,
        MIN(sal) min_sal,
        MAX(sal) max_sal
    FROM emp
    GROUP BY deptno;
```

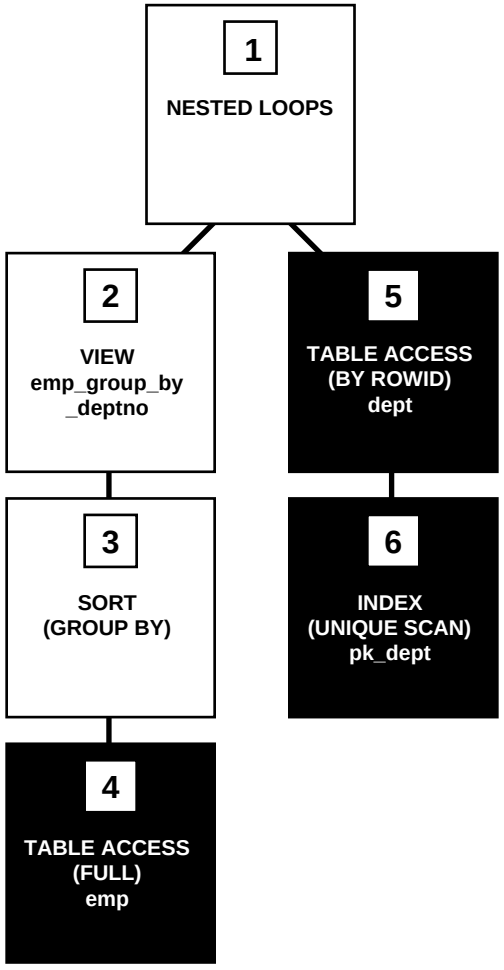
Consider this query, which accesses the view. The query joins the average, minimum, and maximum salaries from each department represented in this view and to the name and location of the department in the DEPT table:

```
SELECT emp_group_by_deptno.deptno, avg_sal, min_sal,
  max_sal, dname, loc
    FROM emp_group_by_deptno, dept
    WHERE emp_group_by_deptno.deptno = dept.deptno;
```

Since there is no equivalent statement that accesses only base tables, the optimizer cannot transform this statement. Instead, the optimizer chooses an execution plan that issues the view's query and then uses the resulting set of rows as it would the rows resulting from a table access.

Figure 20–7, "Joining a View Defined with a GROUP BY Clause to a Table", shows the execution plan for this statement. For more information on how Oracle performs a nested loops join operation, see "Join Operations" on page 20-63.

Figure 20–7 Joining a View Defined with a GROUP BY Clause to a Table



To execute this statement, Oracle performs these operations:

- Step 4 performs a full scan of the EMP table.
- Step 3 sorts the results of Step 4 and calculates the average, minimum, and maximum SAL values selected by the query for the EMP_GROUP_BY_DEPTNO view.
- Step 2 used the data from the previous two steps for a view.

- For each row returned by Step 2, Step 6 uses the DEPTNO value to perform a unique scan of the PK_DEPT index.
- Step 5 uses each ROWID returned by Step 6 to locate the row in the DEPTNO table with the matching DEPTNO value.
- Oracle combines each row returned by Step 2 with the matching row returned by Step 5 and returns the result.

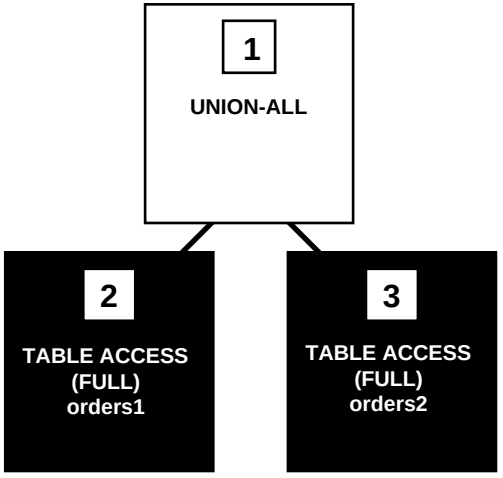
Optimizing Compound Queries

To choose the execution plan for a compound query, the optimizer chooses an execution plan for each of its component queries and then combines the resulting row sources with the union, intersection, or minus operation, depending on the set operator used in the compound query.

Figure 20–8, "Compound Query with UNION ALL Set Operator", shows the execution plan for this statement, which uses the UNION ALL operator to select all occurrences of all parts in either the ORDERS1 table or the ORDERS2 table:

```
SELECT part FROM orders1
UNION ALL
SELECT part FROM orders2;
```

Figure 20–8 Compound Query with UNION ALL Set Operator



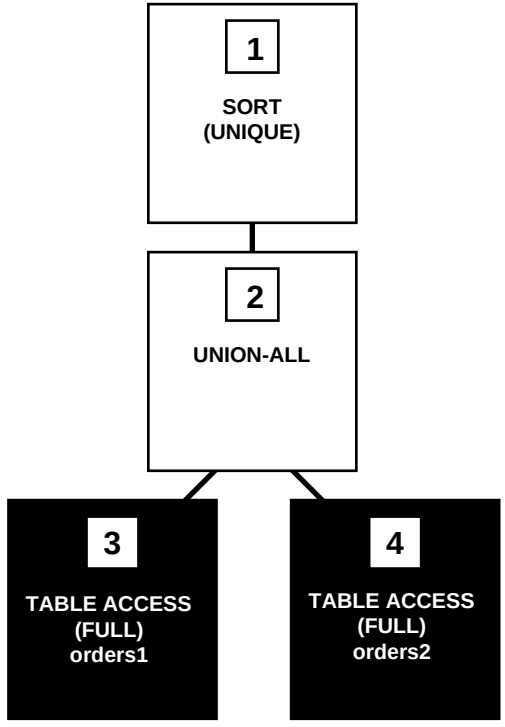
To execute this statement, Oracle performs these steps:

- Steps 2 and 3 perform full table scans on the ORDERS1 and ORDERS2 tables.
- Step 1 performs a UNION-ALL operation returning all rows that are returned by either Step 2 or Step 3 including all copies of duplicates.

Figure 20–9, "Compound Query with UNION Set Operator", shows the execution plan for the following statement, which uses the UNION operator to select all parts that appear in either the ORDERS1 or ORDERS2 table:

```
SELECT part FROM orders1
UNION
SELECT part FROM orders2;
```

Figure 20–9 Compound Query with UNION Set Operator

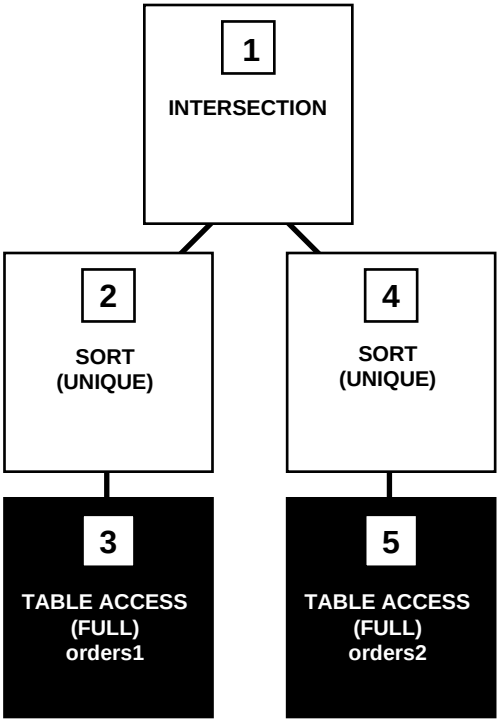


This execution plan is identical to the one for the UNION-ALL operator shown in Figure 20–8 “Compound Query with UNION ALL Set Operator”, except that in this case Oracle uses the SORT operation to eliminate the duplicates returned by the UNION-ALL operation.

Figure 20–10, “Compound Query with INTERSECT Set Operator”, shows the execution plan for this statement, which uses the INTERSECT operator to select only those parts that appear in both the ORDERS1 and ORDERS2 tables:

```
SELECT part FROM orders1
INTERSECT
SELECT part FROM orders2;
```

Figure 20–10 Compound Query with INTERSECT Set Operator



To execute this statement, Oracle performs these steps:

- Steps 3 and 5 perform full table scans of the ORDERS1 and ORDERS2 tables.
- Steps 2 and 4 sort the results of Steps 3 and 5, eliminating duplicates in each row source.
- Step 1 performs an INTERSECTION operation that returns only rows that are returned by both Steps 2 and 4.

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 accesses tables on a single database. Oracle then sends each fragment to the database that it accesses. The remote Oracle instance for each of these databases executes its fragment and returns the results to the local database, where the local Oracle instance may perform any additional processing the statement requires.

When choosing a cost-based execution plan for a distributed statement, the optimizer considers the available indexes on remote databases just as it does indexes on the local database. The optimizer also considers statistics on remote databases for cost-based optimization. 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.

For a rule-based execution plan, the optimizer does not consider indexes on remote tables.

Choosing an Optimization Approach and Goal

The optimizer's behavior when choosing an optimization approach and goal for a SQL statement is affected by these factors:

- the OPTIMIZER_MODE initialization parameter
- statistics in the data dictionary
- the OPTIMIZER_GOAL parameter of the ALTER SESSION command
- hints (comments) in the SQL statement
- the statement being executed in a PL/SQL block

The OPTIMIZER_MODE Initialization Parameter

The OPTIMIZER_MODE initialization parameter establishes the default behavior for choosing an optimization approach for the instance. This parameter can have these values:

CHOOSE	The optimizer chooses between a cost-based approach and a rule-based approach based on whether the statistics are available for the cost-based approach. If the data dictionary contains statistics for at least one of the accessed tables, the optimizer uses a cost-based approach and optimizes with a goal of best throughput. If the data dictionary contains no statistics for any of the accessed tables, the optimizer uses a rule-based approach. This is the default value for the parameter.
ALL_ROWS	The optimizer uses a cost-based approach for all SQL statements in the session regardless of the presence of statistics and optimizes with a goal of best throughput (minimum resource use to complete the entire statement).
FIRST_ROWS	The optimizer uses a cost-based approach for all SQL statements in the session regardless of the presence of statistics and optimizes with a goal of best response time (minimum resource use to return the first row of the result set).
RULE	The optimizer chooses a rule-based approach for all SQL statements issued to the instance regardless of the presence of statistics.

If the optimizer uses the cost-based approach for a SQL statement and some tables accessed by the statement have no statistics, the optimizer uses internal informa-

tion (such as the number of data blocks allocated to these tables) to estimate other statistics for these tables.

Statistics in the Data Dictionary

Oracle stores statistics about columns, tables, clusters, indexes, and partitions in the data dictionary for use by the cost-based optimizer (see "Statistics for the Cost-Based Approach" on page 20-7).

Two options of the ANALYZE command generate statistics:

- COMPUTE STATISTICS generates exact statistics.
- ESTIMATE STATISTICS generates estimations by sampling the data.

Additional Information: See *Oracle8 Tuning* for more information.

The OPTIMIZER_GOAL Parameter of the ALTER SESSION Command

The OPTIMIZER_GOAL parameter of the ALTER SESSION command can override the optimization approach and goal established by the OPTIMIZER_MODE initialization parameter for an individual session.

The value of this parameter affects the optimization of SQL statements issued by stored procedures and functions called during the session, but it does not affect the optimization of recursive SQL statements that Oracle issues during the session. The optimization approach for recursive SQL statements is affected only by the value of the OPTIMIZER_MODE initialization parameter.

The OPTIMIZER_GOAL parameter can have these values:

CHOOSE	The optimizer chooses between a cost-based approach and a rule-based approach based on whether statistics are available for the cost-based approach. If the data dictionary contains statistics for at least one of the accessed tables, the optimizer uses a cost-based approach and optimizes with a goal of best throughput. If the data dictionary contains no statistics for any of the accessed tables, the optimizer uses a rule-based approach.
ALL_ROWS	The optimizer uses a cost-based approach for all SQL statements in the session regardless of the presence of statistics and optimizes with a goal of best throughput (minimum resource use to complete the entire statement).

FIRST_ROWS	The optimizer uses a cost-based approach for all SQL statements in the session regardless of the presence of statistics and optimizes with a goal of best response time (minimum resource use to return the first row of the result set).
RULE	The optimizer chooses a rule-based approach for all SQL statements issued to the Oracle instance regardless of the presence of statistics.

The FIRST_ROWS, ALL_ROWS, CHOOSE, and RULE Hints

A FIRST_ROWS, ALL_ROWS, CHOOSE, or RULE hint in an individual SQL statement can override the effects of both the OPTIMIZER_MODE initialization parameter and the OPTIMIZER_GOAL parameter of the ALTER SESSION command.

> **Additional Information:** See *Oracle8 Tuning* for information on how to use hints.

PL/SQL and the Optimizer Goal

The optimizer goal applies only to queries submitted directly, not queries submitted from within PL/SQL.

- The ALTER SESSION OPTIMIZER_GOAL statement does not affect SQL that is run from within PL/SQL.
- PL/SQL ignores the initialization parameter OPTIMIZER_MODE = FIRST_ROWS.

You can use hints to determine the access path for SQL statements submitted from within PL/SQL.

Choosing Access Paths

One of the most important choices the optimizer makes when formulating an execution plan is how to retrieve 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:

- 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 or 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.

The index contains not only the indexed value, but also the ROWIDs of rows in the table having that value. Therefore, if the statement accesses other columns in addition to the indexed columns, Oracle can find 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 scan	A unique scan of an index returns only a single ROWID. Oracle performs a unique scan only in cases in which a single ROWID is required, rather than many ROWIDs. 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 scan	A range scan of an index can return zero or more ROWIDs depending on how many rows the statement accesses.
full scan	Full index scan is available if a predicate references one of the columns in the index. The predicate does not have to be an index driver. Full scan is also available when there is no predicate if all of the columns in the table referenced in the query are included in the index and at least one of the index columns is not nullable. Full scan can be used to eliminate a sort operation. It reads the blocks singly.
fast full scan	Fast full index scan is an alternative to a full table scan when the index contains all the columns that are needed for the query and at least one column in the index key has the NOT NULL constraint. Fast full scan accesses the data in the index itself, without accessing the table. It cannot be used to eliminate a sort operation. It reads the entire index using multiblock reads (unlike a full index scan) and can be parallelized. Fast full scan is available only with cost-based optimization. You can specify it with the initialization parameter FAST_FULL_SCAN_ENABLED or the INDEX_FFS hint.

bitmap — Bitmap indexes use a bitmap for key values and a mapping function that converts each bit position to a ROWID. Bitmaps efficiently merge indexes that correspond to several conditions in a WHERE clause, using Boolean operations to resolve AND and OR conditions. Bitmap access is available only with cost-based optimization. See "Bitmap Indexes" on page 8-23.

Attention: Bitmap indexes are available only if you have purchased the Oracle8 Enterprise Edition. See *Getting to Know Oracle8 and the Oracle8 Enterprise Edition* for more information.

Access Paths

Table 20–1 lists the data 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.

- The cost-based approach chooses a path based on resource use (see "Choosing Among Access Paths with the Cost-Based Approach" on page 20-58).
- The rule-based approach uses the rank of each path to choose a path when more than one path is available (see "Choosing Among Access Paths with the Rule-Based Approach" on page 20-62).

Table 20–1 Access Paths

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
Unranked Access Paths	
—	Fast full index scan (not available with the rule-based optimizer): see *Oracle8 Tuning*
—	Bitmap index scan (not available with the rule-based optimizer): see “Bitmap Indexes” on page 8-23

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

Path 1: Single Row by ROWID

This access path is available only if the statement's WHERE clause identifies the selected rows by ROWID or with the CURRENT OF CURSOR embedded SQL syntax supported by the Oracle Precompilers. To execute the statement, Oracle accesses the table by ROWID.

Example: This access path is available in the following statement:

```
SELECT * FROM emp WHERE ROWID = 'AAAA7bAA5AAAA1UAAA';
```

The EXPLAIN PLAN output for this statement might look like this:

```
OPERATION                 OPTIONS        OBJECT_NAME
-------------------------------------------------------
SELECT STATEMENT
    TABLE ACCESS          BY ROWID       EMP
```

Path 2: Single Row by Cluster Join

This access path is available for statements that join tables stored in the same cluster if both of these conditions are true:

- The statement's WHERE clause contains conditions that equate each column of the cluster key in one table with the corresponding column in the other table.
- The statement's WHERE clause also contains a condition that guarantees that the join returns only one row. Such a condition is likely to be an equality condition on the column(s) of a unique or primary key.

These conditions must be combined with AND operators. To execute the statement, Oracle performs a nested loops operation. (For information on the nested loops operation, see "Join Operations" on page 20-63.)

Example: This access path is available for the following statement in which the EMP and DEPT tables are clustered on the DEPTNO column and the EMPNO column is the primary key of the EMP table:

```
SELECT *
  FROM emp, dept
  WHERE emp.deptno = dept.deptno
    AND emp.empno = 7900;
```

The EXPLAIN PLAN output for this statement might look like this:

```
OPERATION                 OPTIONS        OBJECT_NAME
--------------------------------------------------
SELECT STATEMENT
    NESTED LOOPS
       TABLE ACCESS       BY ROWID       EMP
       INDEX              UNIQUE SCAN    PK_EMP
       TABLE ACCESS       CLUSTER        DEPT
```

PK_EMP is the name of an index that enforces the primary key.

Path 3: Single Row by Hash Cluster Key with Unique or Primary Key

This access path is available if both of these conditions are true:

- The statement's WHERE clause uses all columns of a hash cluster key in equality conditions. For composite cluster keys, the equality conditions must be combined with AND operators.
- The statement is guaranteed to return only one row because the columns that make up the hash cluster key also make up a unique or primary key.

To execute the statement, Oracle applies the cluster's hash function to the hash cluster key value specified in the statement to obtain a hash value. Oracle then uses the hash value to perform a hash scan on the table.

Example: This access path is available in the following statement in which the ORDERS and LINE_ITEMS tables are stored in a hash cluster, and the ORDERNO column is both the cluster key and the primary key of the ORDERS table:

```
SELECT *
    FROM orders
    WHERE orderno = 65118968;
```

The EXPLAIN PLAN output for this statement might look like this:

```
OPERATION                 OPTIONS        OBJECT_NAME
--------------------------------------------------
SELECT STATEMENT
    TABLE ACCESS          HASH           ORDERS
```

Path 4: Single Row by Unique or Primary Key

This access path is available if the statement's WHERE clause uses all columns of a unique or primary key in equality conditions. For composite keys, the equality conditions must be combined with AND operators. To execute the statement, Oracle

performs a unique scan on the index on the unique or primary key to retrieve a single ROWID and then accesses the table by that ROWID.

Example: This access path is available in the following statement in which the EMPNO column is the primary key of the EMP table:

```
SELECT *
    FROM emp
    WHERE empno = 7900;
```

The EXPLAIN PLAN output for this statement might look like this:

```
OPERATION                 OPTIONS        OBJECT_NAME
-------------------------------------------------------
SELECT STATEMENT
  TABLE ACCESS            BY ROWID       EMP
    INDEX                 UNIQUE SCAN    PK_EMP
```

PK_EMP is the name of the index that enforces the primary key.

Path 5: Clustered Join

This access path is available for statements that join tables stored in the same cluster if the statement's WHERE clause contains conditions that equate each column of the cluster key in one table with the corresponding column in the other table. For a composite cluster key, the equality conditions must be combined with AND operators. To execute the statement, Oracle performs a nested loops operation. (For information on nested loops operations, see "Join Operations" on page 20-63.)

Example: This access path is available in the following statement in which the EMP and DEPT tables are clustered on the DEPTNO column:

```
SELECT *
    FROM emp, dept
    WHERE emp.deptno = dept.deptno;
```

The EXPLAIN PLAN output for this statement might look like this:

```
OPERATION                 OPTIONS        OBJECT_NAME
-------------------------------------------------------
SELECT STATEMENT
  NESTED LOOPS
    TABLE ACCESS          FULL           DEPT
    TABLE ACCESS          CLUSTER        EMP
```

Path 6: Hash Cluster Key

This access path is available if the statement's WHERE clause uses all the columns of a hash cluster key in equality conditions. For a composite cluster key, the equality conditions must be combined with AND operators. To execute the statement, Oracle applies the cluster's hash function to the hash cluster key value specified in the statement to obtain a hash value. Oracle then uses this hash value to perform a hash scan on the table.

Example: This access path is available for the following statement in which the ORDERS and LINE_ITEMS tables are stored in a hash cluster and the ORDERNO column is the cluster key:

```
SELECT *
    FROM line_items
    WHERE orderno = 65118968;
```

The EXPLAIN PLAN output for this statement might look like this:

```
OPERATION                 OPTIONS        OBJECT_NAME
-------------------------------------------------------
SELECT STATEMENT
    TABLE ACCESS          HASH           LINE_ITEMS
```

Path 7: Indexed Cluster Key

This access path is available if the statement's WHERE clause uses all the columns of an indexed cluster key in equality conditions. For a composite cluster key, the equality conditions must be combined with AND operators. To execute the statement, Oracle performs a unique scan on the cluster index to retrieve the ROWID of one row with the specified cluster key value. Oracle then uses that ROWID to access the table with a cluster scan. Since all rows with the same cluster key value are stored together, the cluster scan requires only a single ROWID to find them all.

Example: This access path is available in the following statement in which the EMP table is stored in an indexed cluster and the DEPTNO column is the cluster key:

```
SELECT *  FROM emp
  WHERE deptno = 10;
```

The EXPLAIN PLAN output for this statement might look like this:

```
OPERATION                 OPTIONS        OBJECT_NAME
-------------------------------------------------------
SELECT STATEMENT
  TABLE ACCESS            CLUSTER        EMP
    INDEX                 UNIQUE SCAN    PERS_INDEX
```

PERS_INDEX is the name of the cluster index.

Path 8: Composite Index

This access path is available if the statement's WHERE clause uses all columns of a composite index in equality conditions combined with AND operators. To execute the statement, Oracle performs a range scan on the index to retrieve ROWIDs of the selected rows and then accesses the table by those ROWIDs.

Example: This access path is available in the following statement in which there is a composite index on the JOB and DEPTNO columns:

```
SELECT *
    FROM emp
    WHERE job = 'CLERK'
      AND deptno = 30;
```

The EXPLAIN PLAN output for this statement might look like this:

```
OPERATION                 OPTIONS        OBJECT_NAME
-------------------------------------------------------
SELECT STATEMENT
  TABLE ACCESS            BY ROWID       EMP
    INDEX                 RANGE SCAN     JOB_DEPTNO_INDEX
```

JOB_DEPTNO_INDEX is the name of the composite index on the JOB and DEPTNO columns.

Path 9: Single-Column Indexes

This access path is available if the statement's WHERE clause uses the columns of one or more single-column indexes in equality conditions. For multiple single-column indexes, the conditions must be combined with AND operators.

If the WHERE clause uses the column of only one index, Oracle executes the statement by performing a range scan on the index to retrieve the ROWIDs of the selected rows and then accessing the table by these ROWIDs.

Example: This access path is available in the following statement in which there is an index on the JOB column of the EMP table:

```
SELECT *
    FROM emp
    WHERE job = 'ANALYST';
```

The EXPLAIN PLAN output for this statement might look like this:

```
OPERATION                 OPTIONS        OBJECT_NAME
-------------------------------------------------
SELECT STATEMENT
  TABLE ACCESS            BY ROWID       EMP
    INDEX                 RANGE SCAN     JOB_INDEX
```

JOB_INDEX is the index on EMP.JOB.

If the WHERE clauses uses columns of many single-column indexes, Oracle executes the statement by performing a range scan on each index to retrieve the ROWIDs of the rows that satisfy each condition. Oracle then merges the sets of ROWIDs to obtain a set of ROWIDs of rows that satisfy all conditions. Oracle then accesses the table using these ROWIDs.

Oracle can merge up to five indexes. If the WHERE clause uses columns of more than five single-column indexes, Oracle merges five of them, accesses the table by ROWID, and then tests the resulting rows to determine whether they satisfy the remaining conditions before returning them.

Example: This access path is available in the following statement in which there are indexes on both the JOB and DEPTNO columns of the EMP table:

```
SELECT *
    FROM emp
    WHERE job = 'ANALYST'
      AND deptno = 20;
```

The EXPLAIN PLAN output for this statement might look like this:

```
OPERATION                 OPTIONS        OBJECT_NAME
-------------------------------------------------
SELECT STATEMENT
  TABLE ACCESS            BY ROWID       EMP
    AND-EQUAL
      INDEX               RANGE SCAN     JOB_INDEX
      INDEX               RANGE SCAN     DEPTNO_INDEX
```

The AND-EQUAL operation merges the ROWIDs obtained by the scans of the JOB_INDEX and the DEPTNO_INDEX, resulting in a set of ROWIDs of rows that satisfy the query.

Path 10: Bounded Range Search on Indexed Columns

This access path is available if the statement's WHERE clause contains a condition that uses either the column of a single-column index or one or more columns that make up a leading portion of a composite index:

```
column = expr

column >[=] expr AND column <[=] expr

column BETWEEN expr AND expr

column LIKE 'c%'
```

Each of these conditions specifies a bounded range of indexed values that are accessed by the statement. The range is said to be bounded because the conditions specify both its least value and its greatest value. To execute such a statement, Oracle performs a range scan on the index and then accesses the table by ROWID.

This access path is not available if the expression *expr* references the indexed column.

Example: This access path is available in this statement in which there is an index on the SAL column of the EMP table:

```
SELECT *
    FROM emp
    WHERE sal BETWEEN 2000 AND 3000;
```

The EXPLAIN PLAN output for this statement might look like this:

```
OPERATION                 OPTIONS        OBJECT_NAME
--------------------------------------------------
SELECT STATEMENT
  TABLE ACCESS            BY ROWID       EMP
    INDEX                 RANGE SCAN     SAL_INDEX
```

SAL_INDEX is the name of the index on EMP.SAL.

Example: This access path is also available in the following statement in which there is an index on the ENAME column of the EMP table:

```
SELECT *
    FROM emp
    WHERE ename LIKE 'S%';
```

Path 11: Unbounded Range Search on Indexed Columns

This access path is available if the statement's WHERE clause contains one of these conditions that use either the column of a single-column index or one or more columns of a leading portion of a composite index:

```
WHERE column >[=] expr

WHERE column <[=] expr
```

Each of these conditions specifies an unbounded range of index values accessed by the statement. The range is said to be unbounded because the condition specifies either its least value or its greatest value, but not both. To execute such a statement, Oracle performs a range scan on the index and then accesses the table by ROWID.

Example: This access path is available in the following statement in which there is an index on the SAL column of the EMP table:

```
SELECT *
    FROM emp
    WHERE sal > 2000;
```

The EXPLAIN PLAN output for this statement might look like this:

```
OPERATION                 OPTIONS        OBJECT_NAME
-------------------------------------------------------
SELECT STATEMENT
  TABLE ACCESS            BY ROWID       EMP
    INDEX                 RANGE SCAN     SAL_INDEX
```

Example: This access path is available in the following statement in which there is a composite index on the ORDER and LINE columns of the LINE_ITEMS table:

```
SELECT *
    FROM line_items
    WHERE order > 65118968;
```

The access path is available because the WHERE clause uses the ORDER column, a leading portion of the index.

Example: This access path is not available in the following statement in which there is an index on the ORDER and LINE columns:

```
SELECT *
    FROM line_items
    WHERE line < 4;
```

The access path is not available because the WHERE clause only uses the LINE column, which is not a leading portion of the index.

Path 12: Sort-Merge Join

This access path is available for statements that join tables that are not stored together in a cluster if the statement's WHERE clause uses columns from each table in equality conditions. To execute such a statement, Oracle uses a sort-merge operation. Oracle can also use a nested loops operation to execute a join statement. (For information on these operations, see "Optimizing Join Statements" on page 20-63.)

Example: This access path is available for the following statement in which the EMP and DEPT tables are not stored in the same cluster:

```
SELECT *
    FROM emp, dept
    WHERE emp.deptno = dept.deptno;
```

The EXPLAIN PLAN output for this statement might look like this:

```
OPERATION                 OPTIONS        OBJECT_NAME
-------------------------------------------------------
SELECT STATEMENT
  MERGE JOIN
    SORT                  JOIN
      TABLE ACCESS        FULL           EMP
    SORT                  JOIN
  TABLE ACCESS            FULL           DEPT
```

Path 13: MAX or MIN of Indexed Column

This access path is available for a SELECT statement for which all of these conditions are true:

- The query uses the MAX or MIN function to select the maximum or minimum value of either the column of a single-column index or the leading column of a composite index. The index cannot be a cluster index. The argument to the MAX or MIN function can be any expression involving the column, a constant,

or the addition operator (+), the concatenation operation (||), or the CONCAT function.

- There are no other expressions in the select list.
- The statement has no WHERE clause or GROUP BY clause.

To execute the query, Oracle performs a range scan of the index to find the maximum or minimum indexed value. Since only this value is selected, Oracle need not access the table after scanning the index.

Example: This access path is available for the following statement in which there is an index on the SAL column of the EMP table:

```
SELECT MAX(sal) FROM emp;
```

The EXPLAIN PLAN output for this statement might look like this:

```
OPERATION                 OPTIONS        OBJECT_NAME
-------------------------------------------------------
SELECT STATEMENT
  AGGREGATE               GROUP BY
    INDEX                 RANGE SCAN     SAL_INDEX
```

Path 14: ORDER BY on Indexed Column

This access path is available for a SELECT statement for which all of these conditions are true:

- The query contains an ORDER BY clause that uses either the column of a single-column index or a leading portion of a composite index. The index cannot be a cluster index.
- There must be a PRIMARY KEY or NOT NULL integrity constraint that guarantees that at least one of the indexed columns listed in the ORDER BY clause contains no nulls.
- The NLS_SORT parameter is set to BINARY.

To execute the query, Oracle performs a range scan of the index to retrieve the ROWIDs of the selected rows in sorted order. Oracle then accesses the table by these ROWIDs.

Example: This access path is available for the following statement in which there is a primary key on the EMPNO column of the EMP table:

```
SELECT *
    FROM emp
    ORDER BY empno;
```

The EXPLAIN PLAN output for this statement might look like this:

```
OPERATION                 OPTIONS        OBJECT_NAME
-------------------------------------------------------
SELECT STATEMENT
  TABLE ACCESS            BY ROWID       EMP
    INDEX                 RANGE SCAN     PK_EMP
```

PK_EMP is the name of the index that enforces the primary key. The primary key ensures that the column does not contain nulls.

Path 15: Full Table Scan

This access path is available for any SQL statement, regardless of its WHERE clause conditions.

Note that these conditions make index access paths unavailable:

- column1 > column2
- column1 < column2
- column1 >= column2
- column1 <= column2

where *column1* and *column2* are in the same table.

- column IS NULL
- column IS NOT NULL
- column NOT IN
- column != expr
- column LIKE '%pattern'

regardless of whether *column* is indexed.

- expr = expr2

where *expr* is an expression that operates on a column with an operator or function,

regardless of whether the column is indexed.

- NOT EXISTS subquery
- ROWNUM pseudocolumn in a view
- any condition involving a column that is not indexed

Any SQL statement that contains only these constructs and no others that make index access paths available must use full table scans.

Example: This statement uses a full table scan to access the EMP table:

```
SELECT *
    FROM emp;
```

The EXPLAIN PLAN output for this statement might look like this:

```
OPERATION                 OPTIONS        OBJECT_NAME
-------------------------------------------------------
SELECT STATEMENT
  TABLE ACCESS            FULL           EMP
```

Choosing Among Access Paths

This section describes how the optimizer chooses among available access paths:

- when using the cost-based approach
- when using the rule-based approach

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.

> **Additional Information:** See *Oracle8 Tuning* for information about hints in SQL statements.

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. A query that selects a small percentage of a table's rows has good selectivity, while a query that selects a large percentage of 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 examples below illustrate how 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, which uses an equality condition in its WHERE clause to select all employees named 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 is the number of values for each column in the table.
- USER_TABLES.NUM_ROWS 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, which 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.

Example: Consider this query, which uses a bind variable rather than a literal value for the boundary value in the WHERE clause condition:

```
SELECT *
  FROM emp
  WHERE empno < :e1;
```

The optimizer does not know the value of the bind variable E1. Indeed, the value of E1 may be different for each execution of the query. For this reason, the optimizer cannot use the means described in the previous example to determine selectivity of this query. In this case, the optimizer heuristically guesses a small value for the selectivity of the column (because it is indexed). The optimizer makes this assump-

tion whenever a bind variable is used as a boundary value in a condition with one of the operators <, >, <=, or >=.

The optimizer's treatment of bind variables can cause it to choose different execution plans for SQL statements that differ only in the use of bind variables rather than constants. In one case in which this difference may be especially apparent, the optimizer may choose different execution plans for an embedded SQL statement with a bind variable in an Oracle Precompiler program and the same SQL statement with a constant in SQL*Plus.

Example: Consider this query, which uses two bind variables as boundary values in the condition with the BETWEEN operator:

```
SELECT *
  FROM emp
  WHERE empno BETWEEN :low_e AND :high_e;
```

The optimizer decomposes the BETWEEN condition into these two conditions:

```
empno >= :low_e
empno <= :high_e
```

The optimizer heuristically estimates a small selectiviy for indexed columns in order to favor the use of the index.

Example: Consider this query, which uses the BETWEEN operator to select all employees with employee ID numbers between 7500 and 7800:

```
SELECT *
  FROM emp
  WHERE empno BETWEEN 7500 AND 7800;
```

To determine the selectivity of this query, the optimizer decomposes the WHERE clause condition into these two conditions:

```
empno >= 7500
empno <= 7800
```

The optimizer estimates the individual selectivity of each condition using the means described in a previous example. The optimizer then uses these selectivities (*S1* and *S2*) and the absolute value function (ABS) in this formula to estimate the selectivity (*S*) of the BETWEEN condition:

$S = \text{ABS}(S1 + S2 - 1)$

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 20–1 “Access Paths” on page 20-46

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, which 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.

Optimizing Join Statements

To choose an execution plan for a join statement, the optimizer must make these interrelated decisions:

access paths — As for simple statements, the optimizer must choose an access path to retrieve data from each table in the join statement. (See "Choosing Access Paths" on page 20-42.)

join operations — To join each pair of row sources, Oracle must perform one of these operations:

- nested loops
- sort-merge
- cluster
- hash join (not available with rule-based optimization)

join order — To execute a statement that joins more than two tables, Oracle joins two of the tables, and then joins the resulting row source to the next table. This process is continued until all tables are joined into the result.

Join Operations

The optimizer can use the following operations to join two row sources:

- Nested Loops Join
- Sort-Merge Join
- Cluster Join
- Hash Join

Nested Loops Join

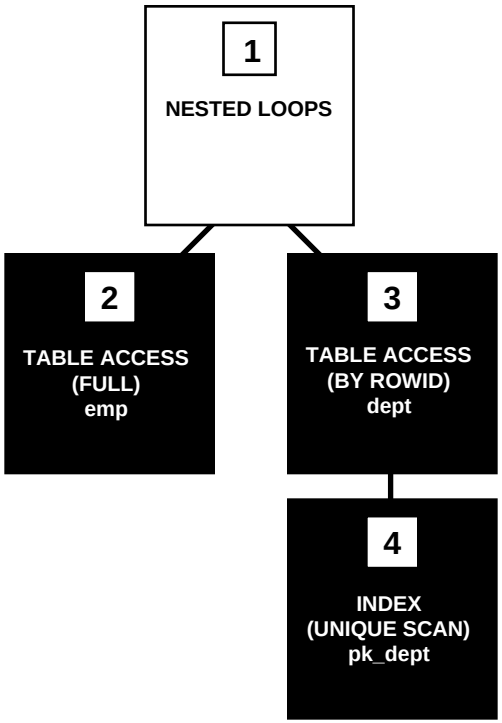
To perform a nested loops join, Oracle follows these steps:

1. The optimizer chooses one of the tables as the *outer table*, or the *driving table*. The other table is called the *inner table*.
2. For each row in the outer table, Oracle finds all rows in the inner table that satisfy the join condition.
3. Oracle combines the data in each pair of rows that satisfy the join condition and returns the resulting rows.

Figure 20–11 shows the execution plan for this statement using a nested loops join:

```
SELECT *
  FROM emp, dept
  WHERE emp.deptno = dept.deptno;
```

Figure 20–11 Nested Loops Join



To execute this statement, Oracle performs these steps:

- Step 2 accesses the outer table (EMP) with a full table scan.
- For each row returned by Step 2, Step 4 uses the EMP.DEPTNO value to perform a unique scan on the PK_DEPT index.
- Step 3 uses the ROWID from Step 4 to locate the matching row in the inner table (DEPT).
- Oracle combines each row returned by Step 2 with the matching row returned by Step 4 and returns the result.

Sort-Merge Join

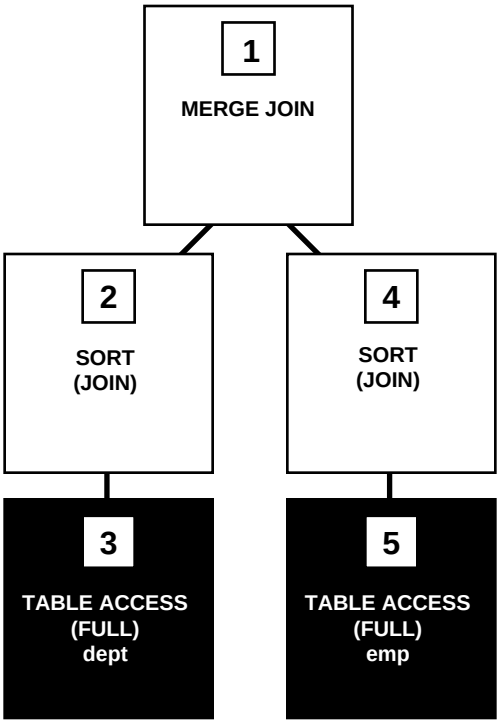
Oracle can only perform a sort-merge join for an equijoin. To perform a sort-merge join, Oracle follows these steps:

1. Oracle sorts each row source to be joined if they have not been sorted already by a previous operation. The rows are sorted on the values of the columns used in the join condition.
2. Oracle merges the two sources so that each pair of rows, one from each source, that contain matching values for the columns used in the join condition are combined and returned as the resulting row source.

Figure 20–12 shows the execution plan for this statement using a sort-merge join:

```
SELECT *
  FROM emp, dept
  WHERE emp.deptno = dept.deptno;
```

Figure 20–12 Sort-Merge Join



To execute this statement, Oracle performs these steps:

- Steps 3 and 5 perform full table scans of the EMP and DEPT tables.
- Steps 2 and 4 sort each row source separately.
- Step 1 merges the sources from Steps 2 and 4 together, combining each row from Step 2 with each matching row from Step 4, and returns the resulting row source.

Cluster Join

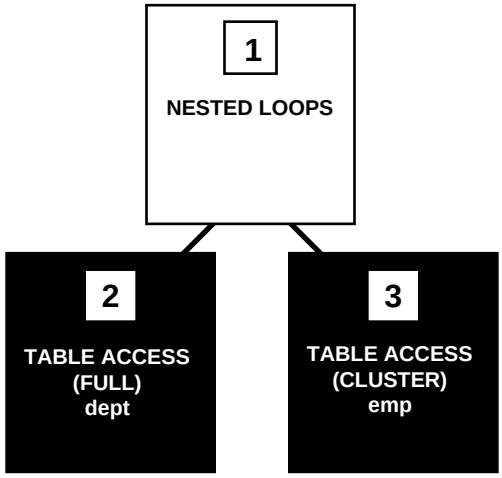
Oracle can perform a cluster join only for an equijoin that equates the cluster key columns of two tables in the same cluster. In a cluster, rows from both tables with the same cluster key values are stored in the same blocks, so Oracle only accesses those blocks.

Additional Information: *Oracle8 Tuning* provides guidelines for deciding which tables to cluster for best performance.

Figure 20–13 shows the execution plan for this statement in which the EMP and DEPT tables are stored together in the same cluster:

```
SELECT *
  FROM emp, dept
  WHERE emp.deptno = dept.deptno;
```

Figure 20–13 Cluster Join



To execute this statement, Oracle performs these steps:

- Step 2 accesses the outer table (DEPT) with a full table scan.
- For each row returned by Step 2, Step 3 uses the DEPT.DEPTNO value to find the matching rows in the inner table (EMP) with a cluster scan.

A cluster join is nothing more than a nested loops join involving two tables that are stored together in a cluster. Since each row from the DEPT table is stored in the same data blocks as the matching rows in the EMP table, Oracle can access matching rows most efficiently.

Hash Join

Oracle can only perform a hash join for an equijoin. Hash join is not available with rule-based optimization. You must enable hash join optimization, using the initialization parameter HASH_JOIN_ENABLED (which can be set with the ALTER SESSION command) or the USE_HASH hint.

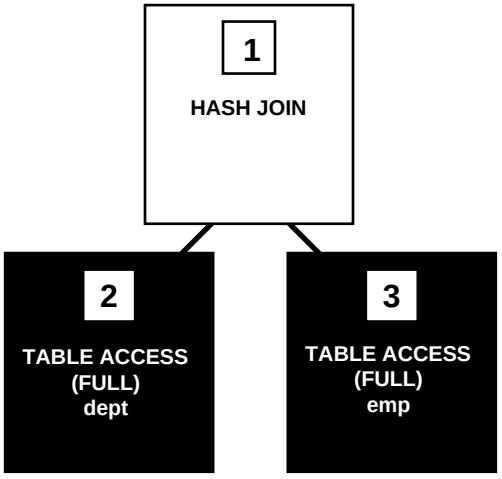
To perform a hash join, Oracle follows these steps:

1. Oracle performs a full table scan on each of the tables and splits each into as many partitions as possible based on the available memory.
2. Oracle builds a hash table from one of the partitions (if possible, Oracle will select a partition that fits into available memory). Oracle then uses the corresponding partition in the other table to probe the hash table. All partition pairs that do not fit into memory are placed onto disk.
3. For each pair of partitions (one from each table), Oracle uses the smaller one to build a hash table and the larger one to probe the hash table.

Figure 20–14 shows the execution plan for this statement using a hash join:

```
SELECT *
  FROM emp, dept
  WHERE emp.deptno = dept.deptno;
```

Figure 20–14 Hash Join



To execute this statement, Oracle performs these steps:

- Steps 2 and 3 perform full table scans of the EMP and DEPT tables.
- Step 1 builds a hash table out of the rows coming from Step 2 and probes it with each row coming from Step 3.

The initialization parameter HASH_AREA_SIZE controls the amount of memory used for hash join operations and the initialization parameter HASH_MULTIBLOCK_IO_COUNT controls the number of blocks a hash join operation should read and write concurrently.

> **Additional Information:** See *Oracle8 Tuning* for more information about these initialization parameters and the USE_HASH hint.

Choosing Execution Plans for Join Statements

This section describes how the optimizer chooses an execution plan for a join statement:

- when using the cost-based approach
- when using the rule-based approach

Note these considerations that apply to the cost-based and rule-based approaches:

- The optimizer first determines whether joining two or more of the tables definitely results in a row source containing at most one row. The optimizer recognizes such situations based on UNIQUE and PRIMARY KEY constraints on the tables. If such a situation exists, the optimizer places these tables first in the join order. The optimizer then optimizes the join of the remaining set of tables.
- For join statements with outer join conditions, the table with the outer join operator must come after the other table in the condition in the join order. The optimizer does not consider join orders that violate this rule.

Choosing Execution Plans for Joins with the Cost-Based Approach

With the cost-based approach, the optimizer generates a set of execution plans based on the possible join orders, join operations, and available access paths. The optimizer then estimates the cost of each plan and chooses the one with the lowest cost. The optimizer estimates costs in these ways:

- The cost of a nested loops operation is based on the cost of reading each selected row of the outer table and each of its matching rows of the inner table into memory. The optimizer estimates these costs using the statistics in the data dictionary.

- The cost of a sort-merge join is based largely on the cost of reading all the sources into memory and sorting them.
- The optimizer also considers other factors when determining the cost of each operation. For example:
  - A smaller sort area size is likely to increase the cost for a sort-merge join because sorting takes more CPU time and I/O in a smaller sort area. Sort area size is specified by the initialization parameter SORT_AREA_SIZE.
  - A larger multiblock read count is likely to decrease the cost for a sort-merge join in relation to a nested loops join. If a large number of sequential blocks can be read from disk in a single I/O, an index on the inner table for the nested loops join is less likely to improve performance over a full table scan. The multiblock read count is specified by the initialization parameter DB_FILE_MULTIBLOCK_READ_COUNT.
  - For join statements with outer join conditions, the table with the outer join operator must come after the other table in the condition in the join order. The optimizer does not consider join orders that violate this rule.

With the cost-based approach, the optimizer's choice of join orders can be overridden with the ORDERED hint. If the ORDERED hint specifies a join order that violates the rule for outer join, the optimizer ignores the hint and chooses the order. You can also override the optimizer's choice of join operations with hints.

> **Additional Information:** See *Oracle8 Tuning* for information on using hints.

Choosing Execution Plans for Joins with the Rule-Based Approach

With the rule-based approach, the optimizer follows these steps to choose an execution plan for a statement that joins R tables:

1. The optimizer generates a set of R join orders, each with a different table as the first table. The optimizer generates each potential join order using this algorithm:
   a. To fill each position in the join order, the optimizer chooses the table with the most highly ranked available access path according to the ranks for access paths in Table 20–1 "Access Paths" on page 20-46. The optimizer repeats this step to fill each subsequent position in the join order.
   b. For each table in the join order, the optimizer also chooses the operation with which to join the table to the previous table or row source in the

order. The optimizer does this by "ranking" the sort-merge operation as access path 12 and applying these rules:

- If the access path for the chosen table is ranked 11 or better, the optimizer chooses a nested loops operation using the previous table or row source in the join order as the outer table.
- If the access path for the table is ranked lower than 12, and there is an equijoin condition between the chosen table and the previous table or row source in join order, the optimizer chooses a sort-merge operation.
- If the access path for the chosen table is ranked lower than 12, and there is not an equijoin condition, the optimizer chooses a nested loops operation with the previous table or row source in the join order as the outer table.

2. The optimizer then chooses among the resulting set of execution plans. The goal of the optimizer's choice is to maximize the number of nested loops join operations in which the inner table is accessed using an index scan. Since a nested loops join involves accessing the inner table many times, an index on the inner table can greatly improve the performance of a nested loops join.

Usually, the optimizer does not consider the order in which tables appear in the FROM clause when choosing an execution plan. The optimizer makes this choice by applying the following rules in order:

a. The optimizer chooses the execution plan with the fewest nested-loops operations in which the inner table is accessed with a full table scan.

b. If there is a tie, the optimizer chooses the execution plan with the fewest sort-merge operations.

c. If there is still a tie, the optimizer chooses the execution plan for which the first table in the join order has the most highly ranked access path:

- If there is a tie among multiple plans whose first tables are accessed by the single-column indexes access path, the optimizer chooses the plan whose first table is accessed with the most merged indexes.
- If there is a tie among multiple plans whose first tables are accessed by bounded range scans, the optimizer chooses the plan whose first table is accessed with the greatest number of leading columns of the composite index.

d. If there is still a tie, the optimizer chooses the execution plan for which the first table appears later in the query's FROM clause.

Views in Outer Joins

For a view that is on the right side of an outer join, the optimzer can use one of two methods, depending on how many base tables the view accesses:

- If the view has only one base table, the optimizer can use *view merging*.
- If the view has multiple base tables, the optimizer can *push the join predicate* into the view.

Merging a View That Has a Single Base Table

A view that has one base table and is on the right side of an outer join can be merged into the query block of an accessing statement. (See "Merging the View's Query into the Statement" on page 20-24.) View merging is possible even if an expression in the view can return a non-null value for a NULL.

Example: Consider the view NAME_VIEW, which concatenates first and last names from the EMP table:

```
CREATE VIEW name_view
  AS SELECT emp.firstname || emp.lastname AS emp_fullname, emp.deptno
       FROM emp;
```

and consider this outer join statement, which finds the names of all employees in London and their departments, as well as any departments that have no employees:

```
SELECT dept.deptno, name_view.emp_fullname
  FROM emp_fullname, dept
  WHERE dept.deptno = name_view.deptno(+)
    AND dept.deptloc = 'London';
```

The optimizer merges the view's query into the outer join statement. The resulting statement looks like this:

```
SELECT dept.deptno, DECODE(emp.rowid, NULL, NULL, emp.firstname || emp.lastname)
  FROM emp, dept
  WHERE dept.deptno = emp.deptno(+)
    AND dept.deptloc = 'London';
```

The transformed statement selects only the employees who work in London.

Pushing the Join Predicate into a View That Has Multiple Base Tables

For a view with multiple base tables on the right side of an outer join, the optimizer can push the join predicate into the view (see "Pushing the Predicate into the View"

on page 20-27) if the initialization parameter PUSH_JOIN_PREDICATE is set to TRUE or the accessing query contains the PUSH_JOIN_PRED hint.

Pushing a join predicate is a cost-based transformation that can enable more efficient access path and join methods, such as transforming hash joins into nested loop joins, and full table scans to index scans.

> **Additional Information:** See *Oracle8 Tuning* for information about optimizer hints.

Example: Consider the view LONDON_EMP, which selects the employees who work in London:

```
CREATE VIEW london_emp
  AS SELECT emp.ename
       FROM emp, dept
       WHERE emp.deptno = dept.deptno
         AND dept.deptloc = 'London';
```

and consider this outer join statement, which finds the engineers and accountants working in London who received bonuses:

```
SELECT bonus.job, london_emp.ename
  FROM bonus, london_emp
  WHERE bonus.job IN ('engineer', 'accountant')
    AND bonus.ename = london_emp.ename(+);
```

The optimizer pushes the outer join predicate into the view. The resulting statement (which does not conform to standard SQL syntax) looks like this:

```
SELECT bonus.job, london_emp.ename
  FROM bonus, (SELECT emp.ename FROM emp, dept
                   WHERE bonus.ename = london_emp.ename(+)
                     AND emp.deptno = dept.deptno
                     AND dept.deptloc = 'London')
  WHERE bonus.job IN ('engineer', 'accountant');
```

Optimizing Anti-Joins and Semi-Joins

An *anti-join* returns rows from the left side of the predicate for which there is no corresponding row on the right side of the predicate. That is, it returns rows that fail to match (NOT IN) the subquery on the right side. For example, an anti-join can select a list of employees who are not in a particular set of departments:

```
SELECT * FROM emp
  WHERE deptno NOT IN
    (SELECT deptno FROM dept
      WHERE loc = 'HEADQUARTERS');
```

The optimizer uses a nested loops algorithm for NOT IN subqueries by default, unless the initialization parameter ALWAYS_ANTI_JOIN is set to MERGE or HASH and various required conditions are met that allow the transformation of the NOT IN subquery into a sort-merge or hash anti-join. You can place a MERGE_AJ or HASH_AJ hint in the NOT IN subquery to specify which algorithm the optimizer should use.

A *semi-join* returns rows that match an EXISTS subquery, without duplicating rows from the left side of the predicate when multiple rows on the right side satisfy the criteria of the subquery. For example:

```
SELECT * FROM dept
  WHERE EXISTS
    (SELECT * FROM emp
      WHERE dept.ename = emp.ename
        AND emp.bonus > 5000);
```

In this query, only one row needs to be returned from DEPT even though many rows in EMP might match the subquery. If there is no index on the BONUS column in EMP, a semi-join can be used to improve query performance.

The optimizer uses a nested loops algorithm for EXISTS subqueries by default, unless the initialization parameter ALWAYS_SEMI_JOIN is set to MERGE or HASH and various required conditions are met. You can place a MERGE_SJ or HASH_SJ hint in the EXISTS subquery to specify which algorithm the optimizer should use.

> **Additional Information:** See *Oracle8 Tuning* for information about optimizer hints.

Optimizing “Star” Queries

One type of data warehouse design centers around what is known as a “star” schema, which is characterized by one or more very large *fact* tables that contain the primary information in the data warehouse and a number of much smaller *dimension* tables (or *“lookup”* tables), each of which contains information about the entries for a particular attribute in the fact table.

A *star query* is a join between a fact table and a number of lookup tables. Each lookup table is joined to the fact table using a primary-key to foreign-key join, but the lookup tables are not joined to each other.

The Oracle cost-based optimizer recognizes star queries and generates efficient execution plans for them. (Star queries are not recognized by the rule-based optimizer.)

A typical fact table contains *keys* and *measures*. For example, a simple fact table might contain the measure Sales, and keys Time, Product, and Market. In this case there would be corresponding dimension tables for Time, Product, and Market. The Product dimension table, for example, would typically contain information about each product number that appears in the fact table.

A *star join* is a primary-key to foreign-key join of the dimension tables to a fact table. The fact table normally has a concatenated index on the key columns to facilitate this type of join.

Star Query Example

This section discusses star queries with reference to the following example:

```
SELECT SUM(dollars)
  FROM facts, time, product, market
  WHERE market.stat = 'New York'
    AND product.brand = 'MyBrand'
    AND time.year = 1995
    AND time.month = 'March'
    /* Joins*/
    AND time.key = facts.tkey
    AND product.pkey = facts.pkey
    AND market.mkey = facts.mkey;
```

Tuning Star Queries

To execute star queries efficiently, you must use the cost based optimizer. Begin by using the ANALYZE command to gather statistics for each of the tables accessed by the query.

Indexing

In the example above, you would construct a concatenated index on the columns tkey, pkey, and mkey. The order of the columns in the index is critical to performance. the columns in the index should take advantage of any ordering of the data. If rows are added to the large table in time order, then tkey should be the first key in the index. When the data is a static extract from another database, it is worthwhile to sort the data on the key columns before loading it.

If all queries specify predicates on each of the small tables, a single concatenated index suffices. If queries that omit leading columns of the concatenated index are frequent, additional indexes may be useful. In this example, if there are frequent queries that omit the time table, an index on pkey and mkey can be added.

Hints

Usually, if you analyze the tables the optimizer will choose an efficient star plan. You can also use hints to improve the plan. The most precise method is to order the tables in the FROM clause in the order of the keys in the index, with the large table last. Then use the following hints:

```
/*+ ORDERED USE_NL(facts) INDEX(facts fact_concat) */
```

A more general method is to use the STAR hint `/*+ STAR */`.

Extended Star Schemas

Each of the small tables can be replaced by a join of several smaller tables. For example, the product table could be normalized into brand and manufacturer tables. Normalization of all of the small tables can cause performance problems. One problem is caused by the increased number of permutations that the optimizer must consider. The other problem is the result of multiple executions of the small table joins. You can solve both of these problems by using denormalized views. For example:

```
CREATE VIEW prodview AS SELECT /*+ NO_MERGE */ *
    FROM brands, mfgrs WHERE brands.mfkey = mfgrs.mfkey;
```

This hint reduces the optimizer’s search space and causes caching of the result of the view.

Star Transformation

The star transformation is a cost-based query transformation aimed at executing star queries efficiently. Whereas the star optimization works well for schemas with

a small number of dimensions and dense fact tables, the star transformation may be considered as an alternative if any of the following holds true:

- The number of dimensions is large.
- The fact table is sparse.
- There are queries where not all dimension tables have constraining predicates.

The star transformation does not rely on computing a Cartesian product of the dimension tables, which makes it better suited for cases where fact table sparsity and/or a large number of dimensions would lead to a large Cartesian product with few rows having actual matches in the fact table. In addition, rather than relying on concatenated indexes, the star transformation is based on combining bitmap indexes on individual fact table columns.

The transformation can thus choose to combine indexes corresponding precisely to the constrained dimensions. There is no need to create many concatenated indexes where the different column orders match different patterns of constrained dimensions in different queries.

Attention: Bitmap indexes are available only if you have purchased the Oracle8 Enterprise Edition. In Oracle8, bitmap indexes are not available and star query processing uses B*-tree indexes. In the Oracle8 Enterprise Edition, the parallel bitmap index join algorithm is also available for star query processing.

See *Getting to Know Oracle8 and the Oracle8 Enterprise Edition* for more information about the features available in Oracle8 and Oracle8 Enterprise Edition.

The star transformation works by generating new subqueries that can be used to drive a bitmap index access path for the fact table.

Consider a simple case with three dimension tables, "d1", "d2", and "d3", and a fact table, "fact". The following query:

```
EXPLAIN PLAN FOR
  SELECT * FROM fact, d1, d2, d3
    WHERE fact.c1 = d1.c1 AND fact.c2 = d2.c1 AND fact.c3 = d3.c1
    AND d1.c2 IN (1, 2, 3, 4)
    AND d2.c2 < 100
    AND d3.c2 = 35
```

gets transformed by adding three subqueries:

```
SELECT * FROM fact, d1, d2
  WHERE fact.c1 = d1.c1 AND fact.c2 = d2.c1
    AND d1.c2 IN (1, 2, 3, 4)
    AND d2.c2 < 100
    AND fact.c1 IN (SELECT d1.c1 FROM d1 WHERE d1.c2 IN (1, 2, 3, 4))
    AND fact.c2 IN (select d2.c1 FROM d2 WHERE d2.c2 < 100)
    AND fact.c3 IN (SELECT d3.c1 FROM d3 WHERE d3.c2 = 35)
```

Given that there are bitmap indexes on fact.c1, fact.c2, and fact.c3, the newly generated subqueries can be used to drive a bitmap index access path in the following way. For each value of d1.c1 that is retrieved from the first subquery, the bitmap for that value is retrieved from the index on fact.c1 and these bitmaps are merged. The result is a bitmap for precisely those rows in fact that match the condition on d1 in the subquery WHERE-clause.

Similarly, the values from the second subquery are used together with the bitmap index on fact.c2 to produce a merged bitmap corresponding to the rows in fact that match the condition on d2 in the second subquery. The same operations apply to the third subquery. The three merged bitmaps can then be ANDed, resulting in a bitmap corresponding to those rows in fact that meet the conditions in all three subqueries simultaneously.

This bitmap can be used to access fact and retrieve the relevant rows. These are then joined to d1, d2, and d3 to produce the answer to the query. No Cartesian product is needed.

Execution Plan

The following execution plan might result from the query above:

```
SELECT STATEMENT
 HASH JOIN
  HASH JOIN
   HASH JOIN
    TABLE ACCESS              FACT      BY ROWID
     BITMAP CONVERSION                  TO ROWIDS
      BITMAP AND
       BITMAP MERGE
        BITMAP KEY ITERATION
         TABLE ACCESS         D3        FULL
         BITMAP INDEX         FACT_C3   RANGE SCAN
       BITMAP MERGE
        BITMAP KEY ITERATION
         TABLE ACCESS         D1        FULL
```

```
        BITMAP INDEX          FACT_C1   RANGE SCAN
      BITMAP MERGE
       BITMAP KEY ITERATION
        TABLE ACCESS          D2        FULL
        BITMAP INDEX          FACT_C2   RANGE SCAN
    TABLE ACCESS              D1        FULL
   TABLE ACCESS               D2        FULL
  TABLE ACCESS                D3        FULL
```

In this plan the fact table is accessed through a bitmap access path based on a bitmap AND of three merged bitmaps. The three bitmaps are generated by the BITMAP MERGE row source being fed bitmaps from row source trees underneath it. Each such row source tree consists of a BITMAP KEY ITERATION row source which fetches values from the subquery row source tree, which in this example is just a full table access, and for each such value retrieves the bitmap from the bitmap index. After the relevant fact table rows have been retrieved using this access path, they are joined with the dimension tables to produce the answer to the query.

The star transformation is a cost-based transformation in the following sense. The optimizer generates and saves the best plan it can produce without the transformation. If the transformation is enabled, the optimizer then tries to apply it to the query and if applicable, generates the best plan using the transformed query. Based on a comparison of the cost estimates between the best plans for the two versions of the query, the optimizer will then decide whether to use the best plan for the transformed or untransformed version.

If the query requires accessing a large percentage of the rows in the fact table, it may well be better to use a full table scan and not use the tranformations. However, if the constraining predicates on the dimension tables are sufficiently selective that only a small portion of the fact table needs to be retrieved, the plan based on the transformation will probably be superior.

Note that the optimizer will generate a subquery for a dimension table only if it decides that it is reasonable to do so based on a number of criteria. There is no guarantee that subqueries will be generated for all dimension tables. The optimizer may also decide, based on the properties of the tables and the query, that the transformation does not merit being applied to a particular query. In this case the best regular plan will be used.

Using Star Transformation

You can enable star transformation by setting the value of the initialization parameter STAR_TRANSFORMATION_ENABLED to TRUE. Use the

STAR_TRANSFORMATION hint to make the optimizer use the best plan in which the transformation has been used.

Restrictions on Star Transformation

Star transformation is not supported for tables with any of the following characteristics:

- tables with a table hint that is incompatible with a bitmap access path
- tables with too few bitmap indexes (there must be a bitmap index on a fact table column for the optimizer to consider generating a subquery for it)
- remote tables (however, remote dimension tables are allowed in the subqueries that are generated)
- anti-joined tables
- tables that are already used as a dimension table in a subquery
- tables that are really unmerged views, which are not view partitions
- tables that have a good single-table access path
- tables that are too small for the transformation to be worthwhile

Part VI

Parallel SQL and Direct-Load INSERT

Part VI describes parallel execution of SQL statements and the direct-load INSERT feature. It contains the following chapters:

- Chapter 21, “Direct-Load INSERT”
- Chapter 22, “Parallel Execution”

21

Direct-Load INSERT

The translator of Homer should above all be penetrated by a sense of four qualities of his author: that he is eminently rapid; that he is eminently plain and direct ... both in his syntax and in his words; that he is eminently plain and direct in the substance of his thought, ...; and, finally, that he is eminently noble.

Matthew Arnold: *On Translating Homer*

This chapter describes the Oracle direct-load INSERT feature for serial or parallel inserts. It also describes the NOLOGGING feature that is available for direct-load INSERT and some DDL statements. This chapter's topics include:

- Introduction to Direct-Load INSERT
- Varieties of Direct-Load INSERT Statements
  - Serial and Parallel INSERT
  - Logging Mode
- Additional Considerations for Direct-Load INSERT
- Restrictions on Direct-Load INSERT

See Chapter 22, "Parallel Execution" for parallel-specific issues.

Attention: The parallel direct-load INSERT feature described in this chapter is available only if you have purchased the Oracle8 Enterprise Edition. See *Getting to Know Oracle8 and the Oracle8 Enterprise Edition* for more information.

Additional Information: See *Oracle8 Tuning* for information on how to tune parallel direct-load INSERT.

Introduction to Direct-Load INSERT

Direct-load INSERT enhances performance during insert operations by formatting and writing data directly into Oracle datafiles, without using the buffer cache. This functionality is similar to that of the Direct Loader utility (SQL*Loader).

Direct-load INSERT appends the inserted data after existing data in a table; free space within the existing data is not reused. Data can be inserted into partitioned or nonpartitioned tables, either in parallel or serially.

Several options of direct-load INSERT exist with respect to parallelism, table partitioning, and logging. "Varieties of Direct-Load INSERT Statements" on page 21-3 describes these features. For additional information about the parallellism and partitioning options of direct-load INSERT, see Chapter 22, "Parallel Execution".

Advantages of Direct-Load INSERT

A major benefit of direct-load INSERT is that you can load data without logging redo or undo entries, which improves the insert performance significantly. Both serial and parallel direct-load INSERT have this performance advantage over conventional path INSERT.

With the conventional path INSERT, in contrast, free space in the object is reused and referential integrity can be maintained. The conventional path for insertions cannot be parallelized.

Comparison with CREATE TABLE ... AS SELECT

With direct-load INSERT, you can insert data into existing tables instead of having to create new tables. Direct-load INSERT updates the indexes of the table, but CREATE TABLE ... AS SELECT only creates a new table which does not have any indexes. See "CREATE TABLE ... AS SELECT in Parallel" on page 22-26 for more information.

Advantage over Parallel Direct Load (SQL*Loader)

With a parallel INSERT, atomicity of the transaction is ensured. Atomicity cannot be guaranteed if multiple parallel loads are used. Also, parallel load could leave some table indexes in an "Unusable" state at the end of the load if errors occurred while updating the indexes. In comparison, parallel INSERT atomically updates the table and indexes (that is, it rolls back the statement if errors occur while updating the index).

> **Additional Information:** See *Oracle8 Utilities* for information about parallel load.

INSERT ... SELECT Statements

Direct-load INSERT (serial or parallel) can only support the INSERT ... SELECT syntax of an INSERT statement, not the INSERT... *values* syntax. The parallelism for INSERT ... SELECT is determined from either parallel hints or parallel table definition clauses.

> **Additional Information:** See *Oracle8 SQL Reference* for information about the syntax of INSERT ... SELECT statements.

Varieties of Direct-Load INSERT Statements

Direct-load INSERT can be performed either:

- serially or in parallel
- into nonpartitioned or partitioned tables
- with or without logging of redo data

Serial and Parallel INSERT

Direct-load INSERT can be done on partitioned or nonpartitioned tables, and it can be done either serially or in parallel.

- **Serial direct-load INSERT into a nonpartitioned or partitioned table.** Data is inserted beyond the current high water mark of the table segment or each partition segment. When a commit executes, the high water mark is updated to the new value, making the data visible to others.
- **Parallel direct-load INSERT into a nonpartitioned table.** Each parallel server process allocates a new temporary segment and inserts data into the temporary segment. When a commit executes, the coordinator process merges the new temporary segments into the primary table segment. (For information about the coordinator process and parallel server processes, see "Process Architecture for Parallel Execution" on page 22-5.)
- **Parallel direct-load INSERT into a partitioned table.** Each parallel server process is assigned one or more partitions, with no more than one process working on any given partition. The parallel server process inserts data beyond the current high water mark of the partition segment(s) assigned to it. When a commit executes, the high water mark of each partition segment is updated by the coordinator process to the new value, making the data visible to others.

In all the cases, the bumping of the high water mark or merging of the temporary segment is delayed until commit is issued, because this action immediately makes the data visible to other processes (that is, it commits the insert operation).

Specifying Serial or Parallel Direct-Load INSERT

The APPEND hint is required for using serial direct-load INSERT. Parallel direct-load INSERT requires either a PARALLEL hint in the statement or a PARALLEL clause in the table definition; the APPEND hint is optional. Parallel direct-load INSERT also requires parallel DML to be enabled with the ALTER SESSION ENABLE PARALLEL DML statement.

Table 21–1 summarizes these requirements and compares direct-load INSERT with conventional INSERT.

Table 21–1 Summary of Serial and Parallel INSERT ... SELECT Statements

Insert Type	Serial	Parallel
Direct-load INSERT	Yes: requires ■ APPEND hint in SQL statement	Yes: requires ■ ALTER SESSION ENABLE PARALLEL DML ■ table PARALLEL attribute or statement PARALLEL hint (an APPEND hint is optional)
Conventional INSERT	Yes (default)	No

Examples of Serial and Parallel Direct-Load INSERT

You can specify serial direct-load INSERT with the APPEND hint, for example:

```
INSERT /*+ APPEND */ INTO emp
   SELECT * FROM t_emp;
COMMIT;
```

You can specify parallel direct-load INSERT by setting the PARALLEL attribute of the table into which rows are inserted, for example:

```
ALTER TABLE emp PARALLEL (DEGREE 10);
ALTER SESSION ENABLE PARALLEL DML;
INSERT INTO emp
   SELECT * FROM t_emp;
COMMIT;
```

You can also specify parallelism for the SELECT operation by setting the PARALLEL attribute of the table from which rows are selected:

```
ALTER TABLE emp PARALLEL (DEGREE 10);
ALTER TABLE t_emp PARALLEL (DEGREE 10);
ALTER SESSION ENABLE PARALLEL DML;
INSERT INTO emp
   SELECT * FROM t_emp;
COMMIT;
```

The PARALLEL hint for an INSERT or SELECT operation takes precedence over a table's PARALLEL attribute. For example, the degree of parallelism in the following INSERT ... SELECT statement is 12 regardless of whether the PARALLEL attributes are set for the EMP and T_EMP tables:

```
ALTER SESSION ENABLE PARALLEL DML;
INSERT /*+ PARALLEL(emp,12) */ INTO emp
   SELECT /*+ PARALLEL(t_emp,12) */ * FROM t_emp;
COMMIT;
```

For more information on parallel INSERT statements, see "Rules for Parallelizing INSERT ... SELECT" on page 22-20.

Logging Mode

Direct-load INSERT operations can be done with or without logging of redo information. You can specify no-logging mode for the table, partition, or index into which data will be inserted by using an ALTER TABLE, ALTER INDEX, or ALTER TABLESPACE command.

- **Direct-load INSERT with logging.** This mode does full redo logging for instance and media recovery. Logging is the default mode.
- **Direct-load INSERT with no-logging.** In this mode, data is inserted without redo or undo logging. (Some minimal logging is still done for marking new extents invalid, and dictionary changes are always fully logged.) When applied during media recovery, the *extent invalidation* records mark a range of blocks as logically corrupt, since the redo data is not logged.

The no-logging mode improves performance because it generates much less log. The user is responsible for backing up the data after a no-logging insert operation in order to be able to perform media recovery.

Note: Logging / no-logging mode is not a permanent attribute of the table, partition, or index. After the database object inserted into has been populated with data and backed up, you can set its status to logging mode so that subsequent changes will be logged.

Table 21–2 compares the LOGGING and NOLOGGING modes for direct-load and conventional INSERT.

Table 21–2 Summary of LOGGING and NOLOGGING Options

Insert Type	LOGGING	NOLOGGING
Direct-load INSERT	Yes: recoverability requires ■ ARCHIVELOG database mode	Yes: requires ■ NOLOGGING attribute for tablespace, table, partition, or index
Conventional INSERT	Yes (default): recoverability requires ■ ARCHIVELOG database mode	No

Examples of No-Logging Mode

You can specify no-logging mode for direct-load INSERT by setting the NOLOGGING attribute of the table into which rows are inserted, for example:

```
ALTER TABLE emp NOLOGGING;
ALTER SESSION ENABLE PARALLEL DML;
INSERT /*+ PARALLEL(emp,12) */ INTO emp
   SELECT /*+ PARALLEL(t_emp,12) */ * FROM t_emp;
COMMIT;
```

You can also set the NOLOGGING attribute for a partition, tablespace, or index; for example:

```
ALTER TABLE emp MODIFY PARTITION emp_lmnop NOLOGGING;

ALTER TABLESPACE personnel NOLOGGING;

ALTER INDEX emp_ix NOLOGGING;

ALTER INDEX emp_ix MODIFY PARTITION eix_lmnop NOLOGGING;
```

SQL Statements That Can Use No-Logging Mode

Although you can set the NOLOGGING attribute for a table, partition, index, or tablespace, no-logging mode does not apply to every operation performed on the schema object for which you set the NOLOGGING attribute.

Only the following operations can make use of no-logging mode:

- direct load (SQL*Loader)
- direct-load INSERT
- CREATE TABLE ... AS SELECT
- CREATE INDEX
- ALTER TABLE ... MOVE PARTITION
- ALTER TABLE ... SPLIT PARTITION
- ALTER INDEX ... SPLIT PARTITION
- ALTER INDEX ... REBUILD
- ALTER INDEX ... REBUILD PARTITION

All of these SQL statements can be parallelized (see Chapter 22, “Parallel Execution”). They can execute in logging or no-logging mode for both serial and parallel execution.

Other SQL statements (such as UPDATE, DELETE, conventional path INSERT, and various DDL statements not listed above) are unaffected by the NOLOGGING attribute of the schema object.

Default Logging Mode

If the LOGGING or NOLOGGING clause is not specified, the logging attribute of the table, partition, or index defaults to the logging attribute of the tablespace in which it resides.

For LOBs, if the LOGGING or NOLOGGING clause is omitted, then:

- if CACHE is specified, LOGGING is used (because LOBs cannot have CACHE NOLOGGING)
- otherwise, the default is obtained from the tablespace in which the LOB value resides.

Additional Considerations for Direct-Load INSERT

This section describes index maintenance, space allocation, and data locks for direct-load INSERT.

Index Maintenance

For direct-load INSERT on nonpartitioned tables or partitioned tables that have local or global indexes, index maintenance is done at the end of the INSERT operation. This index maintenance is performed by the parallel server processes for parallel direct-load INSERT or by the single process for serial direct-load INSERT on partitioned or nonpartitioned tables.

If your direct-load INSERT modifies most of the data in a table, you can avoid the performance impact of index maintenance by dropping the index before the INSERT and then rebuilding it afterwards.

Space Considerations

Direct-load INSERT requires more space than conventional path INSERT, because direct-load INSERT ignores existing space in the free lists of the segment. For parallel direct-load INSERT into nonpartitioned tables, free blocks below the high water mark of the table segment are also ignored. Additional space requirements must be considered before using direct-load INSERT.

Parallel direct-load INSERT into a nonpartitioned table creates temporary segments — one segment for each degree of parallelism. For example, if you use parallel INSERT into a nonpartitioned table with the degree of parallelism set to four, then four temporary segments are created.

Each parallel server process first inserts its data into a temporary segment, and finally the data in all of the temporary segments is appended to the table. (This is the same mechanism as CREATE TABLE ... AS SELECT.)

To provide sufficient storage for temporary segments, without wasting space on segments that are larger than you need, you should specify appropriate values for the storage parameters NEXT and PCTINCREASE for a nonpartitioned table into which you want to do a parallel INSERT. You can change the values of these parameters with the STORAGE option of the ALTER TABLE statement. After performing the parallel DML statement, you can change the NEXT and PCTINCREASE storage parameters back to settings appropriate for non-parallel operations.

Note: The PCTINCREASE storage parameter can produce very large temporary segments, unless it is set to 0. To avoid running out of space while doing parallel DML, make sure that PCTINCREASE is set to 0.

For parallel INSERT into a partitioned table, however, no temporary segments are created. Each parallel server process simply inserts its data into a partition above the high water mark.

Additional Information: Refer to the parallel execution chapter in *Oracle8 Tuning* for more discussion of space management.

Locking Considerations

In direct-load INSERT, exclusive locks are obtained on the table (or on all the partitions of a partitioned table) precluding any concurrent insert, update, or delete on the table. Concurrent queries, however, are supported and will see only the data in the table before the INSERT began. These locks also prevent any concurrent index creation or rebuild operations. This must be taken into account before using direct-load INSERT because it affects table concurrency. For more information, see "Lock and Enqueue Resources for Parallel DML" on page 22-36.

Restrictions on Direct-Load INSERT

The restrictions on direct-load INSERT are the same as those imposed on direct-path parallel loading with SQL*Loader, because they use the same underlying mechanism. In addition, the general parallel DML restrictions also apply to direct-load INSERT.

Serial and parallel direct-load INSERT have the following restrictions:

- A transaction can contain multiple direct-load INSERT statements (or both direct-load INSERT statements and parallel UPDATE or DELETE statements), but after one of these statements modifies a table, no other SQL statement in the transaction can access the same table.
  - Queries that access the same table are allowed before the direct-load INSERT statement, but not after.
  - Any serial or parallel statements attempting to access a table that has already been modified by a direct-load INSERT (or parallel DML) within the same transaction are rejected with an error message.

- If the initialization parameter ROW_LOCKING = INTENT, then inserts cannot be performed by the direct-load path.
- Direct-load INSERT does not support referential integrity.
- Triggers are not supported for direct-load INSERT operations.
- Replication functionality is not supported for direct-load INSERT.
- Direct-load INSERT cannot occur on tables with object columns or LOB columns, or on index-organized tables.
- A transaction involved in a direct-load INSERT operation cannot be or become a distributed transaction.
- Clustered tables are not supported.

Violations of the restrictions will cause the statement to execute serially, using the conventional insert path, without warnings or error messages. An exception is the restriction on statements accessing the same table more than once in a transaction, which can cause error messages.

For example, if triggers or referential integrity are present on the table, then the APPEND hint will be ignored when you try to use direct-load INSERT (serial or parallel), as well as the PARALLEL hint or clause, if any.

For more information about the general restrictions on parallel DML (including parallel INSERT), see “Restrictions on Parallel DML” on page 22-37.

22

Parallel Execution

Civilization advances by extending the number of important operations which we can perform without thinking about them.

Alfred North Whitehead: *An Introduction to Mathematics*

This chapter describes the parallel execution of SQL statements. The topics in this chapter include:

- Overview of Parallel Execution
- Process Architecture for Parallel Execution
- Setting the Degree of Parallelism
- Parallel DDL
- Parallel DML
- Affinity

Attention: The parallel execution features described in this chapter are available only if you have purchased the Oracle8 Enterprise Edition. See *Getting to Know Oracle8 and the Oracle8 Enterprise Edition* for more information about Oracle8 Enterprise Edition.

Note: Parallel execution is not the same as the Oracle Parallel Server (the *Parallel Server option*). You do not need the Parallel Server option to perform parallel execution of SQL statements; however, some aspects of parallel execution apply only to the Oracle Parallel Server.

Overview of Parallel Execution

When Oracle is not parallelizing the execution of SQL statements, each SQL statement is executed sequentially by a single process. With parallel execution, however, multiple processes work together simultaneously to execute a single SQL statement. By dividing the work necessary to execute a statement among multiple processes, Oracle can execute the statement more quickly than if only a single process executed it.

Parallel execution can dramatically improve performance for data-intensive operations associated with decision support applications or very large database environments. Symmetric multiprocessing (SMP), clustered systems, and massively parallel systems (MPP) gain the largest performance benefits from parallel execution because statement processing can be split up among many CPUs on a single Oracle system.

Parallel execution helps systems scale in performance by making optimal use of hardware resources. If your system's CPUs and disk controllers are already heavily loaded, you need to alleviate the system's load or increase these hardware resources before using parallel execution to improve performance.

> **Additional Information:** See *Oracle8 Tuning* for specific information on tuning your parameter files and database to take full advantage of parallel execution.

Operations That Can Be Parallelized

The Oracle server can use parallel execution for any of these operations:

- table scan
- nested loop join
- sort merge join
- hash join
- "not in"
- group by
- select distinct
- union and union all
- aggregation
- PL/SQL functions called from SQL

- order by
- create table as select
- create index
- rebuild index
- rebuild index partition
- move partition
- split partition
- update
- delete
- insert ... select
- enable constraint (the table scan is parallelized)
- star transformation

How Oracle Parallelizes Operations

A SELECT statement only consists of a query. A DML or DDL statement usually consists of a query portion and a DML or DDL portion. Oracle can parallelize both the query portion and the DML or DDL portion of the SQL statements listed in the previous section.

> **Note:** Although generally data manipulation language (DML) includes queries, in this chapter "DML" refers only to inserts, updates, and deletes.

Oracle primarily parallelizes SQL statements in the following ways:

1. Parallelize by block ranges for scan operations (SELECTs and subqueries in DML and DDL statements).
2. Parallelize by partitions for operations on partitioned tables and indexes.
3. Parallelize by parallel server processes for inserts into nonpartitioned tables only.

Parallelizing by Block Range

Oracle parallelizes a query dynamically at execution time. *Dynamic parallelism* divides the table or index into ranges of database blocks *(ROWID range)* and executes the operation in parallel on different ranges. If the distribution or location of data changes, Oracle automatically adapts to optimize the parallelization for each execution of the query portion of a SQL statement.

Parallel scans by block range break the table or index into pieces delimited by high and low ROWID values. The table or index can be nonpartitioned or partitioned.

For partitioned tables and indexes, no ROWID range can span a partition although one partition can contain multiple ROWID ranges. Oracle sends the partition numbers with the ROWID ranges to avoid partition map lookup. Compile and run-time predicates on partitioning columns restrict the ROWID ranges to relevant partitions, eliminating unnecessary partition scans (*partition pruning*).

This means that a parallel query which accesses a partitioned table by a table scan performs the same or less overall work as the same query on a nonpartitioned table. The query on the partitioned table executes with equivalent parallelism, although the total number of disks accessed might be reduced by the partition pruning.

Oracle can parallelize the following operations on tables and indexes by block range (ROWID range):

- queries using table scans (including queries in DML and DDL statements)
- move partition
- split partition
- rebuild index partition
- create index (nonpartitioned index)
- create table ... as select (nonpartitioned table)

Parallelizing by Partition

Partitions are a logical static division of tables and indexes which can be used to break some long-running operations into smaller operations executed in parallel on individual partitions. The granule of parallelism is a partition; there is no parallelism within a partition.

Operations on partitioned tables and indexes are performed in parallel by assigning different parallel server processes to different partitions of the table or index. Compile and run-time predicates restrict the partitions when the operation refer-

ences partitioning columns (partition pruning). The operation executes serially when compile or run-time predicates restrict the operation to a single partition.

The parallel operation may use fewer parallel server processes than the number of accessed partitions (because of resource limits, hints, or table attributes), but each partition is accessed by a single parallel server process. A parallel server process, however, can access multiple partitions.

Operations on partitioned tables and indexes are performed in parallel only when more than one partition is accessed and when the selectivity of the table or index is such that more than a predetermined minimum number of table or index pages will be accessed.

Oracle can parallelize the following operations on partitioned tables and indexes by partition:

- create index
- create table ... as select
- update
- delete
- insert ... select
- alter index ... rebuild
- queries using a range scan on a partitioned index

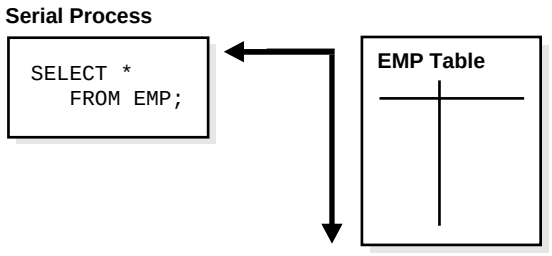
Parallelizing by Parallel Server Processes

For nonpartitioned tables only, Oracle parallelizes insert operations by dividing the work among parallel server processes. Since new rows do not have ROWIDs, the rows are distributed among the parallel server processes to insert them into the free space.

Process Architecture for Parallel Execution

When parallel execution is not being used, a single server process performs all necessary processing for the sequential execution of a SQL statement. For example, to perform a full table scan (such as SELECT * FROM EMP), one process performs the entire operation, as illustrated in Figure 22–1.

Figure 22–1 Serial Full Table Scan



Parallel execution performs thee operations in parallel using multiple *parallel processes*. One process, known as the *parallel coordinator*, dispatches the execution of a statement to several *parallel server processes* and coordinates the results from all of the server processes to send the results back to the user.

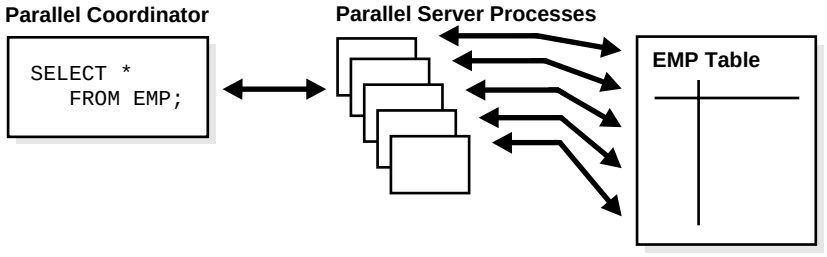
Note: In this context, the phrase "parallel server process" does not mean a process of an Oracle Parallel Server, but instead means a process that performs an operation in parallel. (In an Oracle Parallel Server, however, the parallel server processes may be spread across multiple instances.) Parallel server processes are also sometimes called "slave processes".

When an operation is divided into pieces for parallel execution in a massively parallel processing (MPP) configuration, Oracle assigns a particular piece of the operation to a parallel server process by taking into account the *affinity* of the process for the piece of the table or index to be used for the operation. The physical layout of partitioned tables and indexes impacts on the affinity used to assign work for parallel server processes.

See "Affinity" on page 22-40 for more information.

Figure 22–2 illustrates several parallel server processes simultaneously performing a partial scan of the EMP table, which is divided by block range dynamically *(dynamic partitioning)*. The parallel server processes send results back to the parallel coordinator process, which assembles the pieces into the desired full table scan.

Figure 22–2 Parallel Full Table Scan



The parallel coordinator breaks down execution functions into parallel pieces and then integrates the partial results produced by the parallel server processes. The number of parallel server processes assigned to a single operation is the *degree of parallelism* for an operation. Multiple operations within the same SQL statement all have the same degree of parallelism (see "Determining the Degree of Parallelism for Operations" on page 22-13).

The Parallel Server Pool

When an instance starts up, Oracle creates a pool of parallel server processes which are available for any parallel operation. The initialization parameter PARALLEL_MIN_SERVERS specifies the number of parallel server processes that Oracle creates at instance startup.

When executing a parallel operation, the parallel coordinator obtains parallel server processes from the pool and assigns them to the operation. If necessary, Oracle can create additional parallel server processes for the operation. These parallel server processes remain with the operation throughout job execution, then become available for other operations. After the statement has been processed completely, the parallel server processes return to the pool.

Note: The parallel coordinator and the parallel server processes can only service one statement at a time. A parallel coordinator cannot coordinate, for example, a parallel query and a parallel DML statement at the same time.

You can set PARALLEL_MIN_SERVERS to a higher value if you need to run concurrent parallel statements.

When a user issues a SQL statement, the optimizer decides whether to execute the operations in parallel and determines the degree of parallelism for each operation. You can specify the number of parallel server processes required for an operation in various ways (see "Setting the Degree of Parallelism" on page 22-13).

If the optimizer targets the statement for parallel processing, the following sequence of events takes place:

- The SQL statement's foreground process becomes a parallel coordinator.
- The parallel coordinator obtains as many parallel server processes as needed (determined by the degree of parallelism) from the server pool or creates new parallel server processes as needed.
- Oracle executes the statement as a sequence of operations. Each operation is performed in parallel, if possible.
- When statement processing is completed, the coordinator returns any resulting data to the user process that issued the statement and returns the parallel server processes to the server pool.

The parallel coordinator calls upon the parallel server processes during the execution of the SQL statement (not during the parsing of the statement). Therefore, when parallel execution is used with the multithreaded server, the server process that processes the EXECUTE call of a user's statement becomes the coordinator process for the statement.

Variations in the Number of Parallel Server Processes

If the number of parallel operations processed concurrently by an instance changes significantly, Oracle automatically changes the number of parallel server processes in the pool.

If the number of parallel operations increases, Oracle creates additional parallel server processes to handle incoming requests. However, Oracle never creates more parallel server processes for an instance than what is specified by the initialization parameter PARALLEL_MAX_SERVERS.

If the number of parallel operations decreases, Oracle terminates any parallel server processes that have been idle for the period of time specified by the initialization parameter PARALLEL_SERVER_IDLE_TIME. Oracle does not reduce the size of the pool below the value of PARALLEL_MIN_SERVERS no matter how long the parallel server processes have been idle.

Processing Without Enough Parallel Server Processes

Oracle can process a parallel operation with fewer than the requested number of processes; see "Minimum Number of Parallel Server Processes" on page 22-15 for information about specifying a minimum with the initialization parameter PARALLEL_MIN_PERCENT.

If all parallel server processes in the pool are occupied and the maximum number of parallel server processes has been started, the parallel coordinator switches to serial processing.

> **Additional Information:** See *Oracle8 Tuning* for information about monitoring an instance's pool of parallel server processes and determining the appropriate values of the initialization parameters.

Parallelizing SQL Statements

Each SQL statement undergoes an optimization and parallelization process when it is parsed. Therefore, when the data changes, if a more optimal execution plan or parallelization plan becomes available, Oracle can automatically adapt to the new situation. (Optimization is discussed in Chapter 20, "The Optimizer".)

After the optimizer determines the execution plan of a statement, the parallel coordinator determines the parallelization method for each operation in the execution plan (for example, parallelize a full table scan by block range or parallelize an index range scan by partition). The coordinator must decide whether an operation can be performed in parallel and, if so, how many parallel server processes to enlist (that is, the *degree of parallelism*).

See "Setting the Degree of Parallelism" on page 22-13 and "Parallelization Rules for SQL Statements" on page 22-17 for more information.

Dividing Work Among Parallel Server Processes

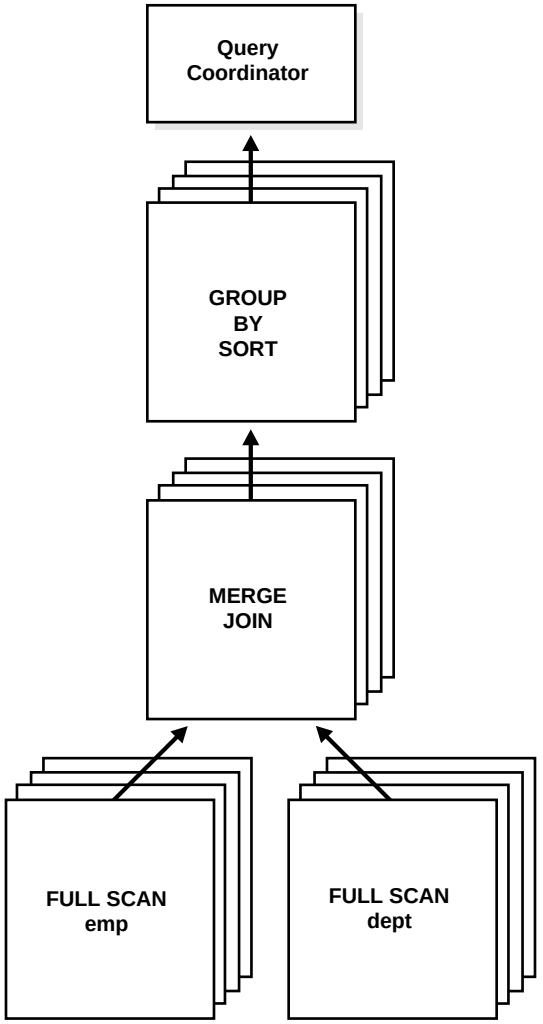
The parallel coordinator examines the redistribution requirements of each operation. An operation's *redistribution requirement* is the way in which the rows operated on by the operation must be divided, or redistributed, among the parallel server processes.

After determining the redistribution requirement for each operation in the execution plan, the optimizer determines the order in which the operations in the execution plan must be performed. With this information, the optimizer determines the data flow of the statement.

Figure 22–3 illustrates the data flow of the following query:

```
SELECT dname, MAX(sal), AVG(sal)
  FROM emp, dept
    WHERE emp.deptno = dept.deptno
    GROUP BY dname;
```

Figure 22–3 Data Flow Diagram for a Join of the EMP and DEPT Tables



Parallelism Between Operations

Operations that require the output of other operations are known as *parent* operations. In Figure 22–3 the GROUP BY SORT operation is the parent of the MERGE JOIN operation because GROUP BY SORT requires the MERGE JOIN output.

Parent operations can begin consuming rows as soon as the child operations have produced rows. In the previous example, while the parallel server processes are producing rows in the FULL SCAN DEPT operation, another set of parallel server processes can begin to perform the MERGE JOIN operation to consume the rows.

Each of the two operations performed concurrently is given its own set of parallel server processes. Therefore, both query operations and the data flow tree itself have parallelism. The parallelism of an individual operation is called *intra-operation* parallelism and the parallelism between operations in a data flow tree is called *inter-operation* parallelism.

Due to the producer/consumer nature of the Oracle server's operations, only two operations in a given tree need to be performed simultaneously to minimize execution time.

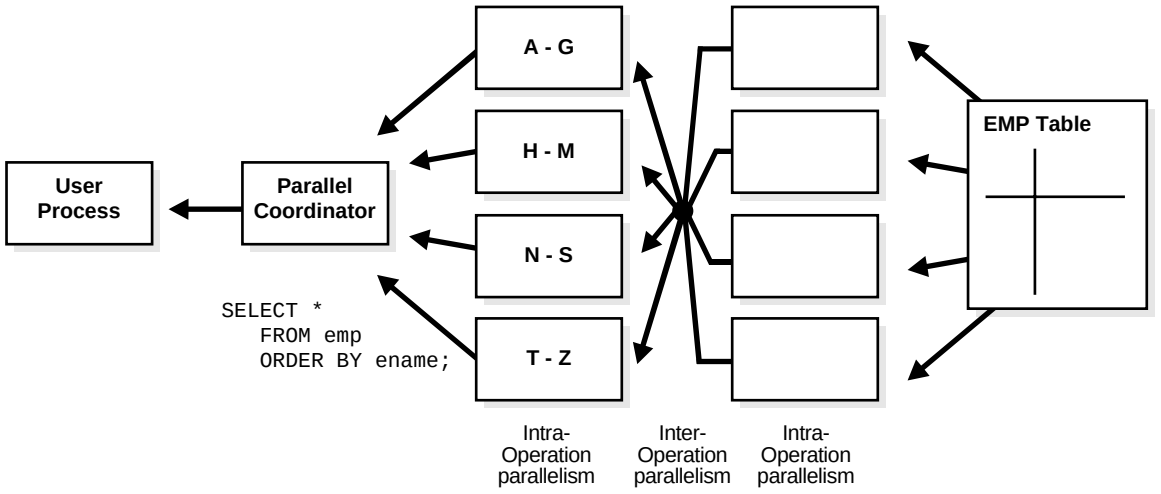
To illustrate intra-operation parallelism and inter-operator parallelism, consider the following statement:

```
SELECT * FROM emp ORDER BY ename;
```

The execution plan consists of a full scan of the EMP table followed by a sorting of the retrieved rows based on the value of the ENAME column. For the sake of this example, assume the ENAME column is not indexed. Also assume that the degree of parallelism for the query is set to four, which means that four parallel server processes can be active for any given operation.

Figure 22–4 illustrates the parallel execution of our example query.

Figure 22–4 Inter-Operation Parallelism and Dynamic Partitioning



As you can see from Figure 22–4, there are actually eight parallel server processes involved in the query even though the degree of parallelism is four. This is because a parent and child operator can be performed at the same time (inter-operation parallelism).

Also note that all of the parallel server processes involved in the scan operation send rows to the appropriate parallel server process performing the sort operation. If a row scanned by a parallel server process contains a value for the ENAME column between A and G, that row gets sent to the first ORDER BY parallel server process. When the scan operation is complete, the sorting processes can return the sorted results to the coordinator, which in turn returns the complete query results to the user.

Note: When a set of parallel server processes completes its operation, it moves on to operations higher in the data flow. For example, in the previous diagram, if there was another ORDER BY operation after the ORDER BY, the parallel server processes performing the table scan perform the second ORDER BY operation after completing the table scan.

Setting the Degree of Parallelism

The parallel coordinator may enlist two or more of the instance's parallel server processes to process a SQL statement. The number of parallel server processes associated with a single operation is known as the *degree of parallelism*.

The degree of parallelism is specified at the statement level (with hints or the PARALLEL clause), at the table or index level (in the table's or index's definition), or by default based on the number of disks or CPUs.

Note that the degree of parallelism applies directly only to intra-operation parallelism. If inter-operation parallelism is possible, the total number of parallel server processes for a statement can be twice the specified degree of parallelism. No more than two operations can be performed simultaneously.

Determining the Degree of Parallelism for Operations

The parallel coordinator determines the degree of parallelism by considering several specifications. The coordinator first checks for hints or a PARALLEL clause specified in the SQL statement itself, then it looks at the table's or index's definition, and finally it checks for the default degree of parallelism (see "Default Degree of Parallelism" on page 22-14). Once a degree of parallelism is found in one of these specifications, it becomes the degree of parallelism for the operation.

For specific details of the degree of parallelism, see "Parallelization Rules for SQL Statements" on page 22-17.

Hints, PARALLEL clauses, table or index definitions, and default values only determine the number of parallel server processes that the coordinator *requests* for a given operation. The actual number of parallel server processes used depends upon how many processes are available in the parallel server pool (see "The Parallel Server Pool" on page 22-7) and whether inter-operation parallelism is possible (see "Parallelism Between Operations" on page 22-11).

Hints

You can specify hints in a SQL statement to set the degree of parallelism for a table or index and the caching behavior of the operation.

- The PARALLEL hint is used only for operations on tables. You can use it to parallelize queries and DML statements (INSERT, UPDATE, and DELETE).
- The PARALLEL_INDEX hint parallelizes an index range scan of a partitioned index. (In an index operation, the PARALLEL hint is not valid and is ignored.)

> **Additional Information:** Refer to *Oracle8 Tuning* for a general discussion on using hints in SQL statements and the specific syntax for the PARALLEL, NOPARALLEL, PARALLEL_INDEX, CACHE, and NOCACHE hints.

Table and Index Definitions

You can specify the degree of parallelism within a table or index definition. Use one of the following SQL statements to set the degree of parallelism for a table or index: CREATE TABLE, ALTER TABLE, CREATE INDEX, or ALTER INDEX.

> **Additional Information:** Refer to the *Oracle8 SQL Reference* for the complete syntax of SQL statements.

Default Degree of Parallelism

The default degree of parallelism is used when you do not specify a degree of parallelism in a hint or within the definition of a table or index. The default degree of parallelism is appropriate for most applications.

> **Additional Information:** See *Oracle8 Tuning* for information about adjusting the degree of parallelism.

The default degree of parallelism for a SQL statement is determined by the following factors.

1. The number of CPUs in the system.
2. The number of Oracle Parallel Server instances.
3. The number of disks (or files, if affinity information is not available) that the table or index is stored on.
4. For parallelizing by partition, the number of partitions that will be accessed, based upon partition pruning (if approximate).
5. For parallel DML operations with global index maintenance, the minimum number of transaction free lists among all the global indexes to be updated. The minimum number of transaction free lists for a partitioned global index is the minimum number across all index partitions. This is a requirement in order to prevent self-deadlock.

For example, if your system has 20 CPUs and you issue a parallel query on a table that is stored on 15 disk drives, then the default degree of parallelism for your query is 15 query servers.

Note: Oracle obtains the information about disks and CPUs from the operating system.

The above factors determine the default number of parallel server processes to use, however, the actual number of processes used is limited by their availability on the requested instances during run time. The initialization parameter PARALLEL_MAX_SERVERS sets an upper limit on the total number of parallel server processes that an instance can have.

If a minimum fraction of the desired parallel server processes is not available (specified by the initialization parameter PARALLEL_MIN_PERCENT), a user error is produced. The user can then retry the query with less parallelism.

Note: The PARALLEL_DEFAULT_SCANSIZE and PARALLEL_DEFAULT_MAX_SCANS initialization parameters are obsolete.

Minimum Number of Parallel Server Processes

Oracle can perform an operation in parallel as long as at least two parallel server processes are available. If too few parallel server processes are available, your SQL statement may execute slower than expected. You can specify that a minimum percentage of requested parallel server processes must be available in order for the operation to execute. This ensures that your SQL statement executes with a minimum acceptable parallel performance. If the minimum percentage of requested parallel server processes are not available, the SQL statement does not execute and returns an error.

The initialization parameter PARALLEL_MIN_PERCENT specifies the desired minimum percentage of requested parallel server processes. This parameter affects DML and DDL operations as well as queries.

For example, if you specify 50 for this parameter, then at least 50% of the parallel server processes requested for any parallel operation must be available in order for the operation to succeed. If 20 parallel server processes are requested, then at least 10 must be available or an error is returned to the user. If PARALLEL_MIN_PERCENT is set to null, then all parallel operations will proceed as long as at least two parallel server processes are available for processing.

Limiting the Number of Available Instances

In an Oracle Parallel Server, instance groups can be used to limit the number of instances that participate in a parallel operation. You can create any number of instance groups, each consisting of one or more instances. You can then specify which instance group is to be used for any or all parallel operations. Parallel server processes will only be used on instances which are members of the specified instance group.

> **Additional Information:** See *Oracle8 Parallel Server Concepts and Administration* for more information about instance groups.

Balancing the Work Load

To optimize performance, all parallel server processes should have equal work loads. For SQL statements parallelized by block range or by parallel server processes, the work load is dynamically divided among the parallel server processes. This minimizes *workload skewing*, which occurs when some parallel server processes perform significantly more work than the other processes.

For SQL statements parallelized by partitions, if the work load is evenly distributed among the partitions then you can optimize performance by matching the number of parallel server processes to the number of partitions, or by choosing a degree of parallelism such that the number of partitions is a multiple of the number of processes.

For example, if a table has ten partitions and a parallel operation divides the work evenly among them, you can use ten parallel server processes (degree of parallelism = 10) to do the work in approximately one-tenth the time that one process would take, or you can use five processes to do the work in one-fifth the time, or two processes to do the work in one-half the time.

If, however, you use nine processes to work on ten partitions, the first process to finish its work on one partition then begins work on the tenth partition; and as the other processes finish their work they become idle. This does not give good performance when the work is evenly divided among partitions. When the work is unevenly divided, the performance varies depending on whether the partition that is left for last has more or less work than the other partitions.

Similarly, if you use four processes to work on ten partitions and the work is evenly divided, then each process works on a second partition after finishing its first partition, but only two of the processes work on a third partition while the other two remain idle.

In general, you cannot assume that the time taken to perform a parallel operation on N partitions with P parallel server processes will be N / P, because of the possibility that some processes might have to wait while others finish working on the last partition(s). By choosing an appropriate degree of parallelism, however, you can minimize the workload skewing and optimize performance.

For information about balancing the work load with disk affinity, see "Affinity and Parallel DML" on page 22-41.

Parallelization Rules for SQL Statements

A SQL statement can be parallelized if it includes a parallel hint or if the table or index being operated on has been declared PARALLEL with a CREATE or ALTER statement. In addition, a data definition language (DDL) statement can be parallelized by using the PARALLEL clause. However, not all of these methods apply to all types of SQL statements.

Parallelization has two components: the decision to parallelize and the degree of parallelism. These components are determined differently for queries, DDL operations, and DML operations.

To determine the degree of parallelism, Oracle looks at the *reference objects*.

- Parallel query looks at each table and index, in the portion of the query being parallelized, to determine which is the reference table. The basic rule is to pick the table or index with the largest degree of parallelism.
- For parallel DML (insert, update, and delete), the reference object that determines the degree of parallelism is the table being modified by an insert, update, or delete operation. Parallel DML also adds some limits to the degree of parallelism to prevent deadlock. If the parallel DML statement includes a subquery, the subquery's degree of parallelism is the same as the DML operation.
- For parallel DDL, the reference object that determines the degree of parallelism is the table, index, or partition being created, rebuilt, split, or moved. If the parallel DDL statement includes a subquery, the subquery's degree of parallelism is the same as the DDL operation.

Rules for Parallelizing Queries

Decision to Parallelize A SELECT statement can be parallelized only if the following conditions are satisfied:

1. The query includes a "parallel" hint specification (PARALLEL or PARALLEL_INDEX) or the schema objects referred to in the query have a PARALLEL declaration associated with them.
2. At least one of the tables specified in the query requires one of the following:
   - a full table scan
   - an index range scan spanning multiple partitions

Degree of Parallelism The degree of parallelism for a query is determined by the following rules:

1. The query uses the maximum degree of parallelism taken from all of the table declarations involved in the query and all of the potential indexes that are candidates to satisfy the query (the *reference objects*). That is, the table or index that has the greatest degree of parallelism determines the query's degree of parallelism (*maximum query directive*).
2. If a table has both a "parallel" hint specification in the query and a parallel declaration in its table specification, the hint specification takes precedence over parallel declaration specification.

Rules for Parallelizing UPDATE and DELETE

Update and delete operations are parallelized by partition. Updates and deletes can only be parallelized on partitioned tables; update/delete parallelism is not possible within a partition, nor on a nonpartitioned table.

You have two ways to specify parallel directives for UPDATE and DELETE operations (assuming that PARALLEL DML mode is enabled):

1. Parallel clause specified in the definition of the table being updated or deleted (the reference object).
2. Update or delete parallel hint specified at the statement.

Parallel hints are placed immediately after the UPDATE or DELETE keywords in UPDATE and DELETE statements. The hint also applies to the underlying scan of the table being changed.

Parallel clauses in CREATE TABLE and ALTER TABLE commands specify table parallelism. If a parallel clause exists in a table definition, it determines the parallelism of DML statements as well as queries. If the DML statement contains explicit parallel hints for a table, however, then those hints override the effect of parallel clauses for that table.

Decision to Parallelize The following rule determines whether the update / delete operation should be parallelized in an update / delete statement:

- The UPDATE or DELETE operation will be parallelized *if and only if t*he table being updated / deleted has a PARALLEL specification or the PARALLEL hint is specified in the DML statement.

If the statement contains subqueries or updatable views, they may have their own separate parallel hints or clauses, but these parallel directives do not affect the decision to parallelize the update or delete.

Although the parallel hint or clause on the tables is used by both query and update / delete portions to determine parallelism, the decision to parallelize the update / delete portion is made independently of the query portion, and vice versa.

Degree of Parallelism The degree of parallelism is determined by the same rules as for the queries. Note that in the case of update and delete operations, only one table (the only reference object) is involved which is the target table to be modified.

The precedence rule to determine the degree of parallelism for the update / delete operation is that the update or delete parallel hint specification takes precedence over the parallel declaration specification of the target table:

Update / Delete hint > Parallel declaration specification of targe table

The maximum degree of parallelism you can achieve is equal to the number of partitions in the table. A parallel server process can update into or delete from multiple partitions, but each partition can only be updated or deleted by one parallel server process.

If the degree of parallelism is less than the number of partitions, then the first process to finish work on one partition continues working on another partition, and so on until the work is finished on all partitions. If the degree of parallelism is greater than the number of partitions involved in the operation, then the excess parallel server processes would have no work to do.

Example 1: `UPDATE tbl_1 SET c1=c1+1 WHERE c1>100;`

If TBL_1 is a partitioned table and its table definition has a parallel clause, then the update operation will be parallelized even if the scan on the table is serial (such as an index scan), assuming that the table has more than one partition with C1 greater than 100.

Example 2: `UPDATE /*+ PARALLEL(tbl_2,4) */ tbl_2 SET c1=c1+1;`

Both the scan and update operations on TBL_2 will be parallelized with degree 4.

Rules for Parallelizing INSERT ... SELECT

An INSERT ... SELECT statement parallelizes its INSERT and SELECT operations independently, except for the degree of parallelism.

You can specify a "parallel" hint after the INSERT keyword in an INSERT ... SELECT statement. Since the tables being queried are usually not the same as the table being inserted into, the hint allows you to specify parallel directives specifically for the insert operation.

You have four ways to specify parallel directives for an INSERT... SELECT statement (assuming that PARALLEL DML mode is enabled):

1. SELECT parallel hint(s) specified at the statement.
2. Parallel clause(s) specified in the definition of tables being selected.
3. INSERT parallel hint specified at the statement.
4. Parallel clause specified in the definition of tables being inserted into.

Decision to Parallelize The following rule determines whether the insert operation should be parallelized in an INSERT... SELECT statement:

- The INSERT operation will be parallelized *if and only if t*he table being inserted into (the reference object) has a PARALLEL declaration specification or the PARALLEL hint is specified after the INSERT in the DML statement.

Hence the decision to parallelize the insert operation is made independently of the select operation, and vice versa.

Degree of Parallelism Once the *decision* to parallelize the select and / or insert operation is made, one parallel directive is picked for deciding *degree* of parallelism of the whole statement using the following precedence rule:

Insert Hint directive > Parallel declaration specification of the inserting table > Maximum Query directive

where *Maximum Query directive* means that among multiple tables and indexes, the table or index that has the maximum degree of parallelism determines the parallelism for the query operation.

The chosen parallel directive is applied to both the select and insert operations.

Example: In the following example, the degree of parallelism used will be 2, which is the degree specified in the Insert hint:

```
INSERT /*+ PARALLEL(tbl_ins,2) */ INTO tbl_ins
  SELECT /*+ PARALLEL(tbl_sel,4) */ * FROM tbl_sel;
```

Rules for Parallelizing DDL Statements

Decision to Parallelize DDL operations can be parallelized if a PARALLEL clause *(declaration)* is specified in the syntax. In the case of CREATE INDEX and ALTER INDEX ... REBUILD or ALTER INDEX ... REBUILD PARTITION, the parallel declaration is stored in the data dictionary.

Degree of Parallelism The degree of parallelism is determined by the specification in the PARALLEL clause. A rebuild of a partitioned index is never parallelized.

Rules for Parallelizing Create Index, Rebuild Index, Merge/Split Partition

Parallel CREATE INDEX or ALTER INDEX ... REBUILD The CREATE INDEX and ALTER INDEX ... REBUILD statements can be parallelized only by a PARALLEL clause.

ALTER INDEX ... REBUILD can be parallelized only for a nonpartitioned index, but ALTER INDEX ... REBUILD PARTITION can be parallelized by a PARALLEL clause.

The scan operation for ALTER INDEX ... REBUILD (nonpartitioned), ALTER INDEX ... REBUILD PARTITION, and CREATE INDEX has the same parallelism as the REBUILD or CREATE operation and uses the same degree of parallelism. If the degree of parallelism is not specified for REBUILD or CREATE, the default is the number of CPUs.

Parallel MOVE PARTITION or SPLIT PARTITION The ALTER INDEX ... MOVE PARTITION and ALTER INDEX ... SPLIT PARTITION statements can be parallelized only by a PARALLEL clause. Their scan operations have the same parallelism as the corresponding MOVE/SPLIT operations. If the degree of parallelism is not specified, the default is the number of CPUs.

Rules for Parellelizing Create Table as Select

The CREATE TABLE ... AS SELECT statement contains two parts:

- a CREATE part (DDL)
- a SELECT part (query)

Oracle can parallelize both parts of the statement. The CREATE part follows the same rules as other DDL operations.

Decision to Parallelize (Query Part) The query part of a CREATE TABLE ... AS SELECT statement can be parallelized only if the following conditions are satisfied:

1. The query includes a "parallel" hint specification (PARALLEL or PARALLEL_INDEX) or the CREATE part of the statement has a PARALLEL clause specification or the schema objects referred to in the query have a PARALLEL declaration associated with them.
2. At least one of the tables specified in the query requires one of the following:
   - a full table scan
   - an index range scan spanning multiple partitions

Degree of Parallelism (Query Part) The degree of parallelism for the query part of a CREATE TABLE ... AS SELECT statement is determined by one of the following rules:

1. The query part uses the values specified in the PARALLEL clause of the CREATE part.
2. If the PARALLEL clause is not specified, the default degree of parallelism is the number of CPUs.

Note that any values specified in a hint for parallelism will be ignored.

Decision to Parallelize (Create Part) The CREATE operation of CREATE TABLE ... AS SELECT can be parallelized only by a PARALLEL clause.

When the CREATE operation of CREATE TABLE ... AS SELECT is parallelized, Oracle also parallelizes the scan operation if possible. The scan operation cannot be parallelized if, for example:

- the SELECT clause has a NOPARALLEL hint
- the operation scans an index of a nonpartitioned table

When the CREATE operation is not parallelized, the SELECT can be parallelized if it has a PARALLEL hint or if the selected table (or partitioned index) has a parallel declaration.

Degree of Parallelism (Create Part) The degree of parallelism for the CREATE operation, and for the SELECT operation if it is parallelized, is specified by the PARALLEL clause of the CREATE statement. If the CREATE statement does not specify the degree of parallelism, the default is the number of CPUs. Note that any degree of parallelism specified in a hint for the SELECT clause is ignored.

Summary of Parallelization Rules

Table 22–1 shows how various types of SQL statements can be parallelized, and indicates which methods of specifying parallelism take precedence.

- The priority (1) specification overrides priority (2) and priority (3).
- The priority (2) specification overrides priority (3).

> **Additional Information:** For details about parallel clauses and hints in SQL statements, see *Oracle8 SQL Reference*.

Table 22–1 Parallelization Rules

Parallel Operation	Parallelized by Clause, Hint, or Underlying Table/Index Declaration (priority order: 1, 2, 3)		
	Parallel Clause	**Parallel Hint**	**Parallel Declaration**
Parallel query table scan (partitioned or nonpartitioned table)		(1) PARALLEL	(2) of table
Parallel query index range scan (partitioned index)		(1) PARALLEL_INDEX	(2) of index
Parallel UPDATE / DELETE (partitioned table only)		(1) PARALLEL	(2) of table being updated or deleted from
Insert operation of parallel INSERT... SELECT (partitioned or nonpartitioned table)		(1) PARALLEL of insert	(2) of table being inserted into
Select operation of parallel INSERT... SELECT (partitioned or nonpartitioned table)		(1) PARALLEL of select	(2) of selecting table
Create operation of parallel CREATE TABLE ... AS SELECT (partitioned or nonpartitioned table)	(1)	(Note: Hint in select clause does not affect the create operation.)	
Select operation of parallel CREATE TABLE ... AS SELECT (partitioned or nonpartitioned table)	(2)	(1) PARALLEL / PARALLEL_INDEX	(3) of querying tables / partitioned indexes
Parallel CREATE INDEX (partitioned or nonpartitioned index)	(1)		
Parallel REBUILD INDEX (nonpartitioned index)	(1)		
REBUILD INDEX (partitioned index)		Never parallelized	
Parallel REBUILD INDEX partition	(1)		
Parallel MOVE / SPLIT partition	(1)		

Parallel DDL

This section includes the following topics on parallelism for data definiation language (DDL) statements:

- DDL Statements That Can Be Parallelized
- CREATE TABLE ... AS SELECT in Parallel
- Recoverability and Parallel DDL
- Space Management for Parallel DDL

DDL Statements That Can Be Parallelized

You can parallelize DDL statements for tables and indexes that are nonpartitioned or partitioned. Table 22–1 on page 22-24 summarizes the operations that can be parallelized in DDL statements.

The parallel DDL statements for nonpartitioned tables and indexes are:

- CREATE INDEX
- CREATE TABLE ... AS SELECT
- ALTER INDEX ... REBUILD

The parallel DDL statements for partitioned tables and indexes are:

- CREATE INDEX
- CREATE TABLE ... AS SELECT
- ALTER TABLE ... MOVE PARTITION
- ALTER TABLE ... SPLIT PARTITION
- ALTER INDEX ... REBUILD PARTITION
- ALTER INDEX ... SPLIT PARTITION — only if the (global) index partition being split is Usable

All of these DDL operations can be performed in no-logging mode (see “Logging Mode” on page 21-5) for either parallel or serial execution.

Different parallelism is used for different operations (see Table 22–1 on page 22-24). Parallel create (partitioned) table as select and parallel create (partitioned) index execute with a degree of parallelism equal to the number of partitions.

Partition parallel analyze table is made less necessary by the ANALYZE {TABLE, INDEX} PARTITION commands, since parallel analyze of an entire partitioned table can be constructed with multiple user sessions.

> **Additional Information:** See *Oracle8 SQL Reference* for information about the syntax and use of parallel DDL statements.

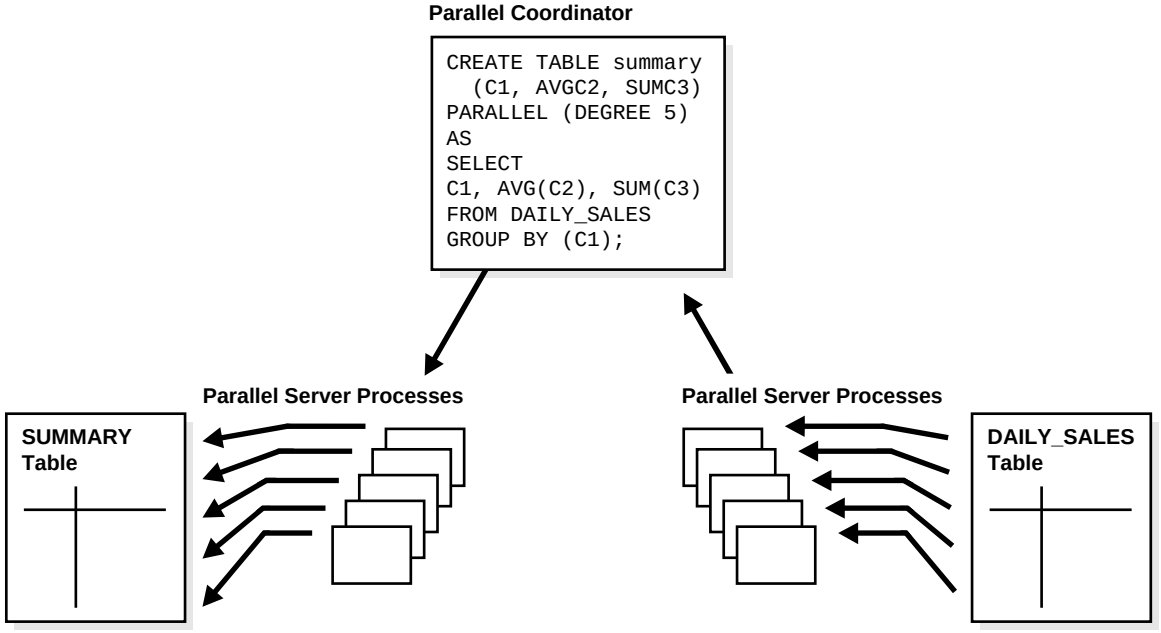
CREATE TABLE ... AS SELECT in Parallel

Decision support applications, for performance reasons, often require large amounts of data to be summarized or "rolled up" into smaller tables for use with ad hoc, decision support queries. Rollup occurs regularly (such as nightly or weekly) during a short period of system inactivity.

Parallel execution allows you to parallelize the query and create operations of creating a table as a subquery from another table or set of tables.

Figure 22–5 illustrates creating a table from a subquery in parallel.

Figure 22–5 Creating a Summary Table in Parallel



Note: Clustered tables cannot be created and populated in parallel.

Recoverability and Parallel DDL

When summary table data is derived from other tables' data, the recoverability from media failure for the smaller summary table may not be important and can be turned off during creation of the summary table.

If you disable logging during parallel table creation (or any other parallel DDL operation), you should take a backup of the tablespace containing the table once the table is created to avoid loss of the table due to media failure.

Use the NOLOGGING clause of CREATE / ALTER TABLE / INDEX statements to disable undo and redo log generation. See "Logging Mode" on page 21-5 for more information.

Additional Information: See the *Oracle8 Administrator's Guide* for information about recoverability of tables created in parallel.

Space Management for Parallel DDL

Creating a table or index in parallel has space management implications that affect both the storage space required during the parallel operation and the free space available after the table or index has been created.

Storage Space for CREATE TABLE ... AS SELECT and CREATE INDEX

When creating a table or index in parallel, each parallel server process uses the values in the STORAGE clause of the CREATE statement to create temporary segments to store the rows. Therefore, a table created with an INITIAL of 5M and a PARALLEL DEGREE of 12 consumes at least 60 megabytes of storage during table creation, because each process starts with an extent of 5 megabytes. When the parallel coordinator combines the segments, some of the segments may be trimmed, and the resulting table may be smaller than the requested 60 megabytes.

Additional Information: See the *Oracle8 SQL Reference* for a discussion of the syntax of the CREATE TABLE command.

Free Space and Parallel DDL

When you create indexes and tables in parallel, each parallel server process allocates a new extent and fills the extent with the table or index's data. Thus, if you

create an index with a degree of parallelism of 3, there will be at least three extents for that index initially. (This discussion also applies to rebuilding indexes in parallel and moving, splitting, or rebuilding partitions in parallel.)

Serial operations require the schema object to have at least one extent. Parallel creations require that tables or indexes have at least as many extents as there are parallel server processes creating the schema object.

When you create a table or index in parallel, it is possible to create "pockets" of free space — either external or internal fragmentation. This occurs when the temporary segments used by the parallel server processes are larger than what is needed to store the rows.

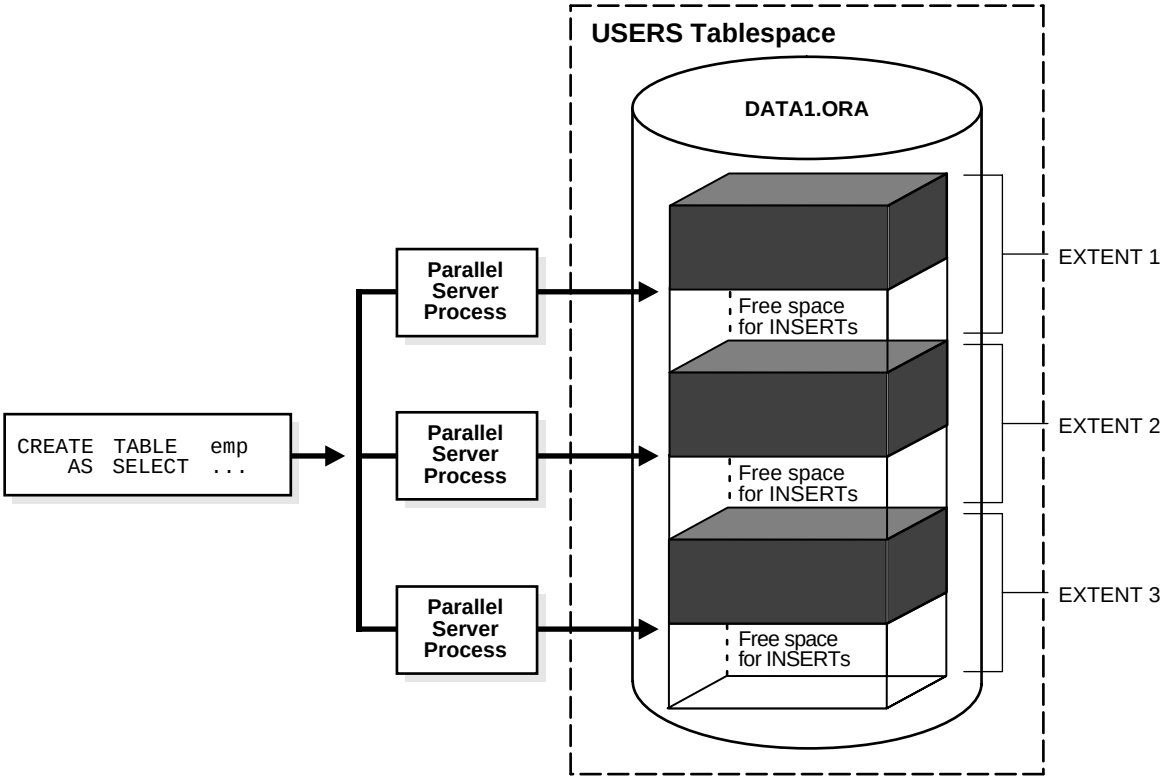
- If the unused space in each temporary segment is larger than the value of the MINIMUM EXTENT parameter set at the tablespace level, Oracle trims the unused space when merging rows from all of the temporary segments into the table or index. The unused space is returned to the system free space and can be allocated for new extents, but it cannot be coalesced into a larger segment because it is not contiguous space (*external fragmentation*).
- If the unused space in each temporary segment is smaller than the value of the MINIMUM EXTENT parameter, unused space cannot be trimmed when the rows in the temporary segments are merged into the table or index. This unused space is not returned to the system free space; it becomes part of the table or index (*internal fragmentation*) and is available only for subsequent inserts or for updates that require additional space.

For example, if you specify a degree of parallelism of three for a CREATE TABLE ... AS SELECT statement but there is only one datafile in the tablespace, the internal fragmentation illustrated in Figure 22–6 can arise. The "pockets" of free space within internal table extents of a datafile cannot be coalesced with other free space and allocated as extents.

For more information about datafiles and tablespaces, see Chapter 3, "Tablespaces and Datafiles".

> **Additional Information:** See *Oracle8 Tuning* for more information about creating tables and indexes in parallel.

Figure 22–6 Unusable Free Space (Internal Fragmentation)



Parallel DML

Parallel DML (parallel insert, update, and delete) uses parallel execution mechanisms to speed up or scale up large DML operations against large database tables and indexes.

Note: Although generally data manipulation language (DML) includes queries, in this chapter the term "DML" refers only to inserts, updates, and deletes.

This section discusses the following parallel DML topics:

- Advantages of Parallel DML over Manual Parallelism
- When to Use Parallel DML
- Enabling Parallel DML
- Transaction Model for Parallel DML
- Recovery for Parallel DML
- Space Considerations for Parallel DML
- Lock and Enqueue Resources for Parallel DML
- Restrictions on Parallel DML

See Chapter 21, "Direct-Load INSERT" for a detailed description of parallel insert statements.

Advantages of Parallel DML over Manual Parallelism

You can parallelize DML operations manually by issuing multiple DML commands simultaneously against different sets of data. For example, you can parallelize manually by:

- issuing multiple INSERT statements to multiple instances of an Oracle Parallel Server to make use of free space from multiple free list blocks
- issuing multiple UPDATE and DELETE statements with different key value ranges or ROWID ranges.

However, manual parallelism has the following disadvantages:

- Difficult to use: you have to open multiple sessions (possibly on different instances) and issue multiple statements.
- Lack of transactional properties: the DML statements are issued at different times, thus the changes are done with inconsistent snapshots of the database. To get atomicity, the commit or rollback of the various statements must be coordinated manually (maybe across instances).
- Work division complexity: you may have to query the table in order to find out the rowid or key value ranges to correctly divide the work.
- Lack of affinity and resource information: you need to know affinity information to issue the right DML statement at the right instance when running an Oracle Parallel Server. You also have to find out about current resource usage to balance work load across instances.

Parallel DML removes these disadvantages by performing inserts, updates, and deletes in parallel automatically.

When to Use Parallel DML

Parallel DML operations are mainly used to speed up large DML operations against large database objects. Parallel DML is useful in a decision support system (DSS) environment where the performance and scalability of accessing large objects are important. Parallel DML complements parallel query in providing you with both querying and updating capabilities for your DSS databases.

The overhead of setting up parallelism makes parallel DML operations infeasible for short OLTP transactions. However, parallel DML operations can speed up batch jobs running in an OLTP database.

Refresh Tables of a Data Warehouse System

In a data warehouse system, large tables need to be *refreshed* (updated) periodically with new or modified data from the production system. You can do this efficiently by using parallel DML combined with updatable join views.

The data that needs to be refreshed is generally loaded into a temporary table before starting the refresh process. This table contains either new rows or rows that have been updated since the last refresh of the data warehouse. You can use an updatable join view with parallel UPDATE to refresh the updated rows, and you can use an anti-hash join with parallel INSERT to refresh the new rows.

Additional Information: For details, see *Oracle8 Tuning*.

Intermediate Summary Tables

In a DSS environment, many applications require complex computations that involve constructing and manipulating many large intermediate summary tables. These summary tables are often temporary and frequently do not need to be logged. Parallel DML can speed up the operations against these large intermediate tables. One benefit is that you can put incremental results in the intermediate tables and perform parallel updates.

In addition, the summary tables may contain cumulative or comparison information which has to persist beyond application sessions; thus, temporary tables are not feasible. Parallel DML operations can speed up the changes to these large summary tables.

Scoring Tables

Many DSS applications score customers periodically based on a set of criteria. The scores are usually stored in large DSS tables. The score information is then used in making a decision, for example, inclusion in a mailing list.

This scoring activity queries and updates a large number of rows in the large table. Parallel DML can speed up the operations against these large tables.

Historical Tables

Historical tables describe the business transactions of an enterprise over a recent time interval. Periodically, the DBA deletes the set of oldest rows and inserts a set of new rows into the table. Parallel INSERT... SELECT and parallel DELETE operations can speed up this rollover task.

Although you can also use parallel direct loader (SQL*Loader) to insert bulk data from an external source, parallel INSERT... SELECT will be faster in inserting data that already exists in another table in the database.

Dropping a partition can also be used to delete old rows, but to do this, the table has to be partitioned by date and with the appropriate time interval.

Batch Jobs

Batch jobs executed in an OLTP database during off hours have a fixed time window in which the jobs must complete. A good way to ensure timely job completion is to parallelize their operations. As the work load increases, more machine resources can be added; the scaleup property of parallel operations ensures that the time constraint can be met.

Enabling Parallel DML

A DML statement can be parallelized only if you have explicitly enabled parallel DML in the session via the ENABLE PARALLEL DML option of the ALTER SESSION statement. This mode is required because parallel DML and serial DML have different locking, transaction, and disk space requirements. (See “Space Considerations for Parallel DML” on page 22-35 and “Lock and Enqueue Resources for Parallel DML” on page 22-36.)

The default mode of a session is DISABLE PARALLEL DML. When PARALLEL DML is disabled, no DML will be executed in parallel even if the PARALLEL hint or PARALLEL clause is used.

When PARALLEL DML is enabled in a session, all DML statements in this session will be *considered* for parallel execution. However, even if the PARALLEL DML is

enabled, the DML operation may still execute serially if there are no parallel hints or parallel clauses or if restrictions on parallel operations are violated (see "Restrictions on Parallel DML" on page 22-37).

The session's PARALLEL DML mode does not influence the parallelism of SELECT statements, DDL statements, and the query portions of DML statements. Thus, if this mode is not set, the DML operation is not parallelized but scans or join operations within the DML statement may still be parallelized.

Transactions with PARALLEL DML Enabled

A session that is enabled for PARALLEL DML may put transactions in the session in a special mode: If any DML statement in a transaction modifies a table in parallel, no subsequent *serial or parallel* query or DML statement can access the same table again in that transaction. This means that the results of parallel modifications cannot be seen during the transaction.

Serial or parallel statements that attempt to access a table which has already been modified in parallel within the same transaction are rejected with an error message.

If a PL/SQL procedure or block is executed in a PARALLEL DML enabled session, then this rule applies to statements in the procedure or block.

Transaction Model for Parallel DML

To execute a DML operation in parallel, the coordinator process acquires or spawns parallel server processes and each parallel server process executes a portion of the work under its own parallel process transaction.

- Each parallel server process creates a different parallel process transaction.
- To reduce contention on the rollback segments, only a few parallel process transactions should reside in the same rollback segment (see the next section).

The coordinator also has its own coordinator transaction, which can have its own rollback segment.

Rollback Segments

Oracle assigns transactions to rollback segments that have the fewest active transactions. To speed up both forward and undo operations, you should create and bring online enough rollback segments so that at most two parallel process transactions are assigned to one rollback segment.

Create the rollback segments in tablespaces that have enough space for them to extend when necessary and set the MAXEXTENTS storage parameters for the rollback segments to UNLIMITED.

Two-Phase Commit

A parallel DML operation is executed by more than one independent parallel process transaction. In order to ensure user-level transactional atomicity, the coordinator uses a two-phase commit protocol to commit the changes performed by the parallel process transactions.

This two-phase commit protocol is a simplified version which makes use of shared disk architecture to speed up transaction status lookups, especially during transactional recovery. It does not require the Oracle XA library. In-doubt transactions never become visible to users.

Recovery for Parallel DML

The time required to roll back a parallel DML operation is roughly equal to the time it took to perform the forward operation.

Oracle supports parallel transaction recovery ("undo" recovery) during transaction and process failures, and to a lesser extent during instance and system failures.

To speed up transaction recovery, the initialization parameter CLEANUP_ROLLBACK_ENTRIES should be set to a high value approximately equal to the number of rollback entries generated for the forward-going operation.

Transaction Recovery

A user-issued rollback in a transaction failure due to statement error is performed in parallel by the parallel coordinator and the parallel server processes. The rollback takes approximately the same amount of time as the forward transaction.

Process Recovery

Recovery from the failure of a parallel DML coordinator or parallel server process is performed by the PMON process.

- If a single parallel server process fails, PMON rolls back that process's work and all other parallel server processes roll back their own work.
- If multiple parallel server processes fail, PMON rolls back all of their work serially.

- If the coordinator process fails, PMON recovers the coordinator and all parallel server processes roll back their own work in parallel.

The recovery time for process failures can therefore be longer than the original (forward) work.

System Recovery

Recovery from a system failure needs a new startup. Recovery is performed by the SMON process. Parallel DML statements recover serially and all resources remain locked until recovery is complete. Recovery can therefore take much longer than the original (forward) transaction if the forward transaction used a high degree of parallelism and has done a lot of work.

One way to speed up transaction recovery is to rerun the parallel DML statement. When the new coordinator and parallel server processes encounter the locked resources, they trigger transaction recovery concurrently. After the new parallel processes finish recovering the resources, you can either commit or roll back the transaction.

Instance Recovery (Oracle Parallel Server)

Recovery from an instance failure in an Oracle Parallel Server is performed by the SMON processes of other live instances. Each SMON process of the live instances can recover the parallel coordinator and / or parallel server process transactions of the failed instance independently. If there are more parallel server processes in the failed instance than there are live instances, the recovery time is longer than the forward work by the failed instance.

Space Considerations for Parallel DML

Parallel UPDATE uses the space in the existing object, as opposed to direct-load INSERT which gets new segments for the data.

Space usage characteristics may be different in parallel than they would be if the statement executed sequentially, because multiple concurrent child transactions modify the object.

See “Space Considerations” on page 21-8 for information about space for direct-load INSERT.

Lock and Enqueue Resources for Parallel DML

A parallel DML operation's lock and enqueue resource requirements are very different from the serial DML requirements. Parallel DML holds many more locks, so you should increase the value of the ENQUEUE_RESOURCES and DML_LOCKS parameters.

The processes for a parallel UPDATE, DELETE, or INSERT statement acquire the following locks:

- The coordinator process acquires:
  - 1 table lock SX
  - 1 partition lock X per partition

  For parallel INSERT into a partitioned table, the coordinator acquires partition locks for all partitions. For parallel UPDATE or DELETE, the coordinator acquires partition locks for all partitions, unless the WHERE clause limits the partitions involved.
- Each parallel server process acquires:
  - 1 table lock SX
  - 1 partition lock NULL per partition
  - 1 partition-wait lock X per partition

A parallel server process can work on one or more partitions, but a partition can only be worked on by one parallel server process.

For example, for a table with 600 partitions running with parallel degree 100, a parallel DML statement needs the following locks (assuming all partitions are involved in the statement):

- The coordinator acquires 1 table lock SX and 600 partition locks X.
- Total parallel server processes acquire 100 table locks SX, 600 partition locks NULL, and 600 partition-wait locks X.

Table 22–2 summarizes the types of locks acquired by coordinator and parallel server processes for different types of parallel DML statements.

Table 22–2 Locks Acquired by Parallel DML Statements

Type of statement	Coordinator process acquires:	Each parallel server process acquires:
Parallel UPDATE or DELETE into partitioned table; WHERE clause specifies the partition	1 table lock SX 1 partition lock X per partition	1 table lock SX 1 partition lock NULL per partition 1 partition-wait lock X per partition
Parallel UPDATE, DELETE, or INSERT into partitioned table	1 table lock SX Partition locks X for all partitions	1 table lock SX 1 partition lock NULL per partition 1 partition-wait lock X per partition
Parallel INSERT into nonpartitioned table	1 table lock X	None

Restrictions on Parallel DML

The following restrictions apply to parallel DML (including direct-load INSERT):

- Update and delete operations are not parallelized on nonpartitioned tables.
- For parallel update operations, global *unique* indexes are not supported. All other indexes are fully maintained by parallel operations.
- A transaction can contain multiple parallel DML statements that modify different tables, but after a parallel DML statement modifies a table, no subsequent *serial or parallel* statement (DML or query) can access the same table again in that transaction.
  - This restriction also exists after a serial direct-load INSERT statement: no subsequent SQL statement (DML or query) can access the modified table during that transaction.
  - Queries that access the same table are allowed before a parallel DML or direct-load INSERT statement, but not after.
  - Any serial or parallel statements attempting to access a table that has already been modified by a parallel UPDATE, parallel DELETE, or direct-load INSERT during the same transaction are rejected with an error message.

- If initialization parameter ROW_LOCKING = INTENT, then inserts, updates, and deletes are ***not*** parallelized (regardless of the serializable mode).
- Triggers are not supported for parallel DML operations.
- Replication functionality is not supported for parallel DML.
- Parallel DML cannot occur in the presence of certain constraints: self-referential integrity, delete cascade, and deferred integrity. In addition, for direct-load INSERT there is no support for any referential integrity.
- Parallel DML cannot occur on tables with object columns or LOB columns, or on index-organized tables.
- A transaction involved in a parallel DML operation cannot be or become a distributed transaction.
- Clustered tables are not supported.

Violations will cause the statement to execute serially without warnings or error messages (except for the restriction on statements accessing the same table in a transaction, which can cause error messages). For example, an update will be serialized if it requires global unique index maintenance.

The following sections give further details about restrictions.

Partitioning Key Restriction

You can only update the partitioning key of a partitioned table to a new value if the update would not cause the row to move to a new partition. This is a general restriction on partitioned tables.

Data Integrity Restrictions

This section describes the interactions of integrity constraints and parallel DML statements.

NOT NULL and CHECK These types of integrity constraints are allowed. They are not a problem for parallel DML because they are enforced on the column and row level, respectively.

UNIQUE and PRIMARY KEY Both of these constraints are enforced with unique indexes. ***An UPDATE command that modifies a unique or primary key index is parallelized only if the index is local.***

FOREIGN KEY (Referential Integrity) There are restrictions for referential integrity whenever a DML operation on one table could cause a recursive DML operation on another table or, in order to perform the integrity check, it would be necessary to see simultaneously all changes made to the object being modified.

Table 22–3 lists all of the operations that are possible on tables that are involved in referential integrity constraints.

Table 22–3 Referential Integrity Restrictions

DML Statement	Issued on Parent	Issued on Child	Self-Referential
INSERT	(Not applicable)	Not parallelized	Not parallelized
UPDATE No Action	Supported	Supported	Not parallelized
DELETE No Action	Supported	Supported	Not parallelized
DELETE Cascade	Not parallelized	(Not applicable)	Not parallelized

Delete Cascade ***Delete on tables having a foreign key with delete cascade is not parallelized*** because parallel server processes will try to delete rows from multiple partitions (parent and child tables).

Self-Referential Integrity ***DML on tables with self-referential integrity constraints is not parallelized if the referenced keys (primary keys) are involved.*** For DML on all other columns, parallelism is possible.

Deferrable Integrity Constraints If there are any deferrable constraints on the table being operated on, the DML operation will not be parallelized.

Trigger Restrictions

A DML operation will not be parallelized if any triggers are enabled on the affected tables that may get fired as a result of the statement. This implies that DML statements on tables that are being replicated will not be parallelized.

Relevant triggers must be disabled in order to parallelize DML on the table. Note that enabling / disabling triggers invalidates dependent shared cursors.

Function Restrictions

Only functions that don't read or write database or package state are allowed in parallel DML statements. A DML operation will not be parallelized if the DML statement has embedded functions that either read or write database state or package state.

Additional Information: See the description of the pragma RESTRICT_REFERENCES in the *Oracle8 Application Developer's Guide*.

Distributed Transaction Restrictions

A DML operation cannot be parallelized if it is in a distributed transaction or if the DML or the query operation is against a remote object.

Example 1: In a distributed transaction:

```
select * from t1@dblink; /* this starts a distributed transaction */
delete /*+ parallel (t2,2) */ from t2; /* not parallelized */
commit;
```

Example 2: DML operation on a remote object:

```
delete /*+ parallel *t1, 2) */ from t1@dblink;
/* cannot parallel delete from remote object */
```

Example 3: DML statement which queries a remote object:

```
insert /* append parallel (t3,2) */ into t3 select * from t4@dblink;
/* not parallelized because of reference to remote object */
```

Affinity

In a shared-disk cluster or massively parallel processing (MPP) configuration, an instance of the Oracle Parallel Server is said to have *affinity* for a device if the device is directly accessed from the processor(s) on which the instance is running. Similarly, an instance has *affinity* for a file if it has affinity for the device(s) that the file is stored on.

Attention: The features described in this section are available only if you have purchased Oracle8 Enterprise Edition with the Parallel Server Option. See *Getting to Know Oracle8 and the Oracle8 Enterprise Edition* for information about the features and options available with Oracle8 Enterprise Edition.

Determination of affinity may involve arbitrary determinations for files that are striped across multiple devices. Somewhat arbitrarily, an instance is said to have affinity for a tablespace (or a partition of a table or index within a tablespace) if the instance has affinity for the first file in the tablespace.

Oracle considers affinity when allocating work to parallel server processes. The use of affinity for parallel execution of SQL statements is transparent to users.

Affinity and Parallel Queries

Affinity in parallel queries increases the speed of scanning data from disk by doing the scans on a processor that is "near" the data. This can provide a substantial performance increase for machines that do not naturally support shared disks.

The most common use of affinity is for a table or index partition to be stored in one file on one device. This configuration provides the highest availability by limiting the damage done by a device failure and makes best use of partition-parallel index scans.

DSS customers might prefer to stripe table partitions over multiple devices (probably a subset of the total number of devices). This allows some queries to prune the total amount of data being accessed using partitioning criteria and still obtain parallelism through ROWID-range parallel table (partition) scans. If the devices are configured as a RAID, availability can still be very good. Even when used for DSS, indexes should probably be partitioned on individual devices.

Other configurations (for example, multiple partitions in one file striped over multiple devices) will yield correct query results, but you may need to use hints or explicitly set object attributes to select the correct degree of parallelism.

Affinity and Parallel DML

For parallel DML (inserts, updates, and deletes), affinity enhancements improve cache performance by routing the DML operation to the node that has affinity for the partition.

Affinity determines how to distribute the work among the set of instances and/or parallel server processes to perform the DML operation in parallel. Affinity can improve performance of queries in several ways:

1. For certain MPP architectures, Oracle uses device-to-node affinity information to determine on which nodes to spawn parallel server processes *(parallel process allocation)* and which work granules (ROWID ranges or partitions) to send to particular nodes *(work assignment)*. Better performance is achieved by having

nodes mainly access local devices, giving a better buffer cache hit ratio for every node and reducing the network overhead and I/O latency.

2. For SMP shared disk clusters, Oracle uses a round-robin mechanism to assign devices to nodes. Similar to item 1, this device-to-node affinity is used in determining parallel process allocation and work assignment.
3. For SMP, cluster, and MPP architectures, process-to-device affinity is used to achieve device isolation. This reduces the chances of having multiple parallel server processes accessing the same device simultaneously. This process-to-device affinity information is also used in implementing stealing between processes.

For partitioned tables and indexes, partition-to-node affinity information determines process allocation and work assignment. For shared-nothing MPP systems, the Oracle Parallel Server tries to assign partitions to instances taking the disk affinity of the partitions into account. For shared-disk MPP and cluster systems, partitions are assigned to instances in a round-robin manner.

Affinity is only available for parallel DML when running in an Oracle Parallel Server configuration. Affinity information which persists across statements will improve buffer cache hit ratios and reduce block pings between instances.

> **Additional Informaion:** See *Oracle8 Parallel Server Concepts and Administration* for more information about the Oracle Parallel Server.

Other Types of Parallelism

In addition to parallel SQL execution, Oracle can use parallelism for the following types of operations:

- parallel recovery
- parallel propagation (replication)
- parallel load (the SQL*Loader utility)

Like parallel SQL, parallel recovery and parallel propagation are executed by a parallel coordinator process and multiple parallel server processes. Parallel load, however, uses a different mechanism.

> **Additional Information:** See *Oracle8 Utilities* for information about parallel load and general information about SQL*Loader. Also see *Oracle8 Tuning* for advice about using parallel load.

The behavior of the parallel coordinator and parallel server processes may differ, depending on what kind of operation they perform (SQL, recovery, or propagation). For example, if all parallel server processes in the pool are occupied and the maximum number of parallel server processes has been started:

- in the parallel SQL role, the parallel coordinator switches to serial processing
- in the parallel propagation role, the parallel coordinator returns an error.

For a given session, the parallel coordinator coordinates only one kind of operation. A parallel coordinator cannot coordinate, for example, parallel SQL and parallel propagation or parallel recovery at the same time.

See “Performing Recovery in Parallel” on page 28-13 for general information about parallel recovery.

> **Additional Information:** See *Oracle8 Backup and Recovery Guide* for detailed information about parallel recovery, and see *Oracle8 Replication* for information about parallel propagation.

Part VII

Data Protection

Part VII describes how Oracle protects the data in a database and explains what the database administrator can do to provide additional protection for data.

Part VII contains the following chapters:

- Chapter 23, “Data Concurrency and Consistency”
- Chapter 24, “Data Integrity”
- Chapter 25, “Controlling Database Access”
- Chapter 26, “Privileges and Roles”
- Chapter 27, “Auditing”
- Chapter 28, “Database Recovery”

23

Data Concurrency and Consistency

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 multiuser database environment. The chapter includes:

- Data Concurrency and Consistency in a Multiuser Environment
- How Oracle Manages Data Concurrency and Consistency
- How Oracle Locks Data

Data Concurrency and Consistency in a Multiuser Environment

In a single-user database, the user can modify data in the database without concern for other users modifying the same data at the same time. However, in a multiuser database, the statements within multiple simultaneous transactions can update the same data. Transactions executing at the same time need to produce meaningful and consistent results. Therefore, control of data concurrency and data consistency is vital in a multiuser database.

- *Data concurrency* means that many users can access data at the same time.
- *Data consistency* means that each user sees a consistent view of the data, including visible changes made by the user's own transactions and transactions of other users.

Data integrity, which enforces business rules associated with a database, is discussed in Chapter 24, "Data Integrity".

To describe consistent transaction behavior when transactions execute at the same time, database researchers have defined a transaction isolation model called *serializability*. The serializable mode of transaction behavior tries to ensure that transactions execute in such a way that they appear to be executed one at a time, or serially, rather than concurrently.

While this degree of isolation between transactions is generally desirable, running many applications in this mode can seriously compromise application throughput. Complete isolation of concurrently running transactions could mean that one transaction cannot perform an insert into a table being queried by another transaction. In short, real-world considerations usually require a compromise between perfect transaction isolation and performance.

Oracle offers two isolation levels, providing application developers with operational modes that preserve consistency and provide high performance.

Preventable Phenomena and Transaction Isolation Levels

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

The three preventable phenomena are:

dirty read	A transaction reads data that has been written by another transaction that has not been committed yet.

nonrepeatable (fuzzy) read	A transaction rereads data it has previously read and finds that another committed transaction has modified or deleted the data.
phantom read	A transaction reexecutes a query returning a set of rows that satisfies a search condition and finds that another committed transaction has inserted additional rows that satisfy the condition.

SQL92 defines four levels of isolation in terms of the phenomena a transaction running at a particular isolation level is permitted to experience.

Isolation Level	Dirty Read	NonRepeatable Read	Phantom Read
Read uncommitted	Possible	Possible	Possible
Read committed	Not possible	Possible	Possible
Repeatable read	Not possible	Not possible	Possible
Serializable	Not possible	Not possible	Not possible

Oracle offers the read committed and serializable isolation levels, as well as a read-only mode that is not part of SQL92. Read committed is the default and was the only automatic isolation level provided before Oracle Release 7.3. The read committed and serializable isolation levels are discussed more fully in "How Oracle Manages Data Concurrency and Consistency" on page 23-4.

Locking Mechanisms

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

Resources include two general types of objects:

- user objects, such as tables and rows (structures and data)
- system objects not visible to users, such as shared data structures in the memory and data dictionary rows

The various types of locks — data locks, DDL locks, and internal locks — are discussed in "How Oracle Locks Data" on page 23-14.

How Oracle Manages Data Concurrency and Consistency

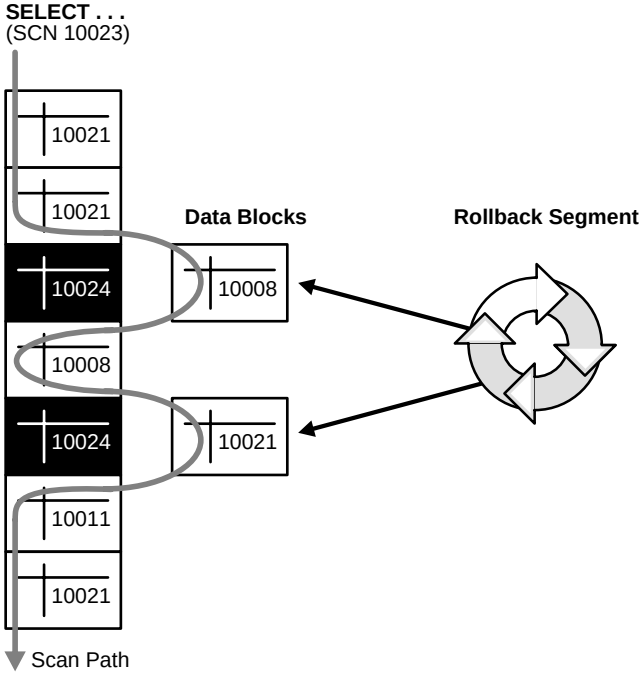
Oracle maintains data consistency in a multiuser environment by using a multiversion consistency model and various types of locks and transactions.

Multiversion Concurrency Control

Oracle automatically provides read consistency to a query so that all the data that the query sees comes from a single point in time (statement-level read consistency). Oracle can also provide read consistency to all of the queries in a transaction (transaction-level read consistency).

Oracle uses the information maintained in its rollback segments to provide these consistent views. The rollback segments contain the old values of data that have been changed by uncommitted or recently committed transactions. Figure 23–1 shows how Oracle provides statement-level read consistency using data in rollback segments.

Figure 23–1 Transactions and Read Consistency



As a query enters the execution stage, the current system change number (SCN) is determined; in Figure 23–1, this system change number is 10023. As data blocks are read on behalf of the query, only blocks written with the observed SCN are used. Blocks with changed data (more recent SCNs) are reconstructed from data in the rollback segments, and the reconstructed data is returned for the query. Therefore, each query returns all *committed* data with respect to the SCN recorded at the time that query execution began. Changes of other transactions that occur during a query's execution are not observed, guaranteeing that consistent data is returned for each query.

The "Snapshot Too Old" Message

In rare situations, Oracle cannot return a consistent set of results (often called a *snapshot*) for a long-running query. This occurs because not enough information remains in the rollback segments to reconstruct the older data. Usually, this error is produced when a lot of update activity causes the rollback segment to wrap around and overwrite changes needed to reconstruct data that the long-running query requires. In this event, error 1555 will result:

```
ORA-1555: snapshot too old (rollback segment too small)
```

You can avoid this error by creating more or larger rollback segments. Alternatively, long-running queries can be issued when there are few concurrent transactions, or you can obtain a shared lock on the table you are querying, thus prohibiting any other exclusive locks during the transaction.

Statement-Level Read Consistency

Oracle always enforces *statement-level* read consistency. This guarantees that all the data returned by a single query comes from a single point in time—the time that the query began. Therefore, a query never sees dirty data nor any of the changes made by transactions that commit during query execution. As query execution proceeds, only data committed before the query began is visible to the query. The query does not see changes committed after statement execution begins.

A consistent result set is provided for every query, guaranteeing data consistency, with no action on the user's part. The SQL statements SELECT, INSERT with a subquery, UPDATE, and DELETE all query data, either explicitly or implicitly, and all return consistent data. Each of these statements uses a query to determine which data it will affect (SELECT, INSERT, UPDATE, or DELETE, respectively).

A SELECT statement is an explicit query and may have nested queries or a join operation. An INSERT statement can use nested queries. UPDATE and DELETE

statements can use WHERE clauses or subqueries to affect only some rows in a table rather than all rows.

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

Transaction-Level Read Consistency

Oracle also offers the option of enforcing *transaction-level read consistency*. When a transaction executes in serializable mode (see below), all data accesses reflect the state of the database as of the time the transaction began. This means that the data seen by all queries within the same transaction is consistent with respect to a single point in time, except that queries made by a serializable transaction do see changes made by the transaction itself. Transaction-level read consistency produces repeatable reads and does not expose a query to phantoms.

Oracle Isolation Levels

Oracle provides three transaction isolation levels:

read committed	This is the default transaction isolation level. Each query executed by a transaction sees only data that was committed before the query (not the transaction) began. An Oracle query will never read dirty (uncommitted) data. Because Oracle does not prevent other transactions from modifying the data read by a query, that data may be changed by other transactions between two executions of the query. Thus, a transaction that executes a given query twice may experience both nonrepeatable read and phantoms.
serializable transactions	Serializable transactions see only those changes that were committed at the time the transaction began, plus those changes made by the transaction itself through INSERT, UPDATE, and DELETE statements. Serializable transactions do not experience nonrepeatable reads or phantoms.
read-only	Read-only transactions see only those changes that were committed at the time the transaction began and do not allow INSERT, UPDATE, and DELETE statements.

Setting the Isolation Level

Application designers, application developers, and database administrators can choose appropriate isolation levels for different transactions, depending on the application and workload. You can set the isolation level of a transaction by using one of these 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;
```

> **Additional Information:** See *Oracle8 SQL Reference* for detailed information on any of these SQL commands.

Read Committed Isolation

The default isolation level for Oracle is read committed. This degree of isolation is appropriate for environments where few transactions are likely to conflict. Oracle causes each query to execute with respect to its own snapshot time, thereby permitting nonrepeatable reads and phantoms for multiple executions of a query, but providing higher potential throughput. Read committed isolation is the appropriate level of isolation for environments where few transactions are likely to conflict.

Serializable Isolation

Serializable isolation is suitable for environments

- with large databases and short transactions that update only a few rows
- where the chance that two concurrent transactions will modify the same rows is relatively low, and
- where relatively long-running transactions are primarily read-only.

Serializable isolation permits concurrent transactions to make only those database changes they could have made if the transactions had been scheduled to execute

one after another. Specifically, Oracle permits a serializable transaction to modify a data row only if it can determine that prior changes to the row were made by transactions that had committed when the serializable transaction began.

To make this determination efficiently, Oracle uses control information stored in the data block that indicates which rows in the block contain committed and uncommitted changes. In a sense, the block contains a recent history of transactions that affected each row in the block. The amount of history that is retained is controlled by the INITRANS parameter of CREATE TABLE and ALTER TABLE.

Under some circumstances, Oracle may have insufficient history information to determine whether a row has been updated by a "too recent" transaction. This can occur when many transactions concurrently modify the same data block, or do so in a very short period. You can avoid this situation by setting higher values of INITRANS for tables that will experience many transactions updating the same blocks. Doing so will enable Oracle to allocate sufficient storage in each block to record the history of recent transactions that accessed the block.

Oracle generates an error when a serializable transaction tries to update or delete data modified by a transaction that commits *after* the serializable transaction began:

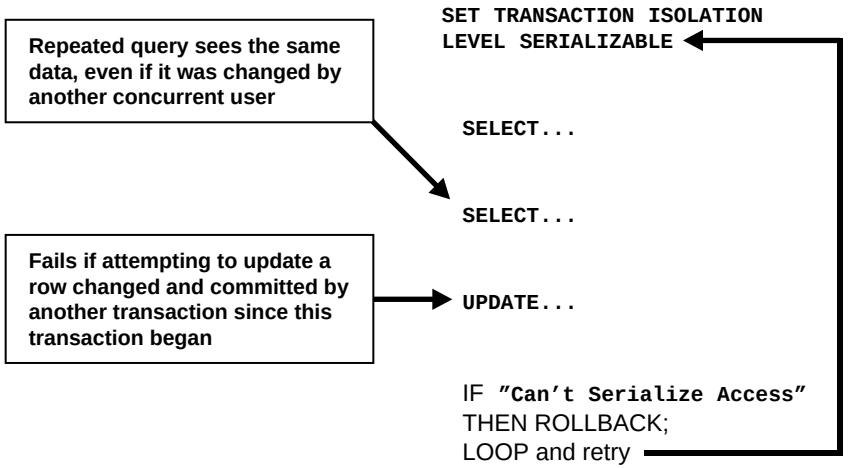
```
ORA-08177: Cannot serialize access for this transaction
```

When a serializable transaction fails with the "Cannot serialize access" error, the application can take any of several actions:

- commit the work executed to that point
- execute additional (but different) statements (perhaps after rolling back to a savepoint established earlier in the transaction)
- roll back the entire transaction

Figure 23–2 shows an example of an application that rolls back and retries the transaction after it fails with the "Cannot serialize access" error:

Figure 23–2 Serializable Transaction Failure



Comparing Read Committed and Serializable Isolation

Oracle gives the application developer a choice of two transaction isolation levels with different characteristics. Both the read committed and serializable isolation levels provide a high degree of consistency and concurrency. Both levels provide the contention-reducing benefits of Oracle's "read consistency" multiversion concurrency control model and exclusive row-level locking implementation and are designed for real-world application deployment.

Transaction Set Consistency

A useful way to view the read committed and serializable isolation levels in Oracle is to consider the following scenario: Assume you have a collection of database tables (or any set of data), a particular sequence of reads of rows in those tables, and the set of transactions committed at any particular time. An operation (a query or a transaction) is *transaction set consistent* if all its reads return data written by the same set of committed transactions. An operation is not transaction set consistent if some reads reflect the changes of one set of transactions and other reads reflect changes made by other transactions. An operation that is not transaction set consistent in effect sees the database in a state that reflects no single set of committed transactions.

Oracle provides transactions executing in read committed mode with transaction set consistency on a per-statement basis. Serializable mode provides transaction set consistency on a per-transaction basis.

Table 23–1 summarizes key differences between read committed and serializable transactions in Oracle.

Table 23–1 Read Committed and Serializable Transactions

	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

Row-Level Locking

Both read committed and serializable transactions use row-level locking, and both will wait if they try to change a row updated by an uncommitted concurrent transaction. The second transaction that tries to update a given row waits for the other transaction to commit or roll back and release its lock. If that other transaction rolls back, the waiting transaction (regardless of its isolation mode) can proceed to change the previously locked row, as if the other transaction had not existed.

However, if the other (blocking) transaction commits and releases its locks, a read committed transaction proceeds with its intended update. A serializable transaction, however, fails with the error "Cannot serialize access", because the other transaction has committed a change that was made since the serializable transaction began.

Referential Integrity

Because Oracle does not use read locks in either read-consistent or serializable transactions, data read by one transaction can be overwritten by another. Transactions that perform database consistency checks at the application level should not assume that the data they read will remain unchanged during the execution of the transaction (even though such changes are not visible to the transaction). Database inconsistencies can result unless such application-level consistency checks are coded with this in mind, even when using serializable transactions.

> **Additional Information:** See *Oracle8 Application Developer's Guide* for more information about referential integrity and serializable transactions.

Oracle Parallel Server

You can use both read committed and serializable transaction isolation levels in an Oracle Parallel Server (several Oracle instances running against a single database).

Distributed Transactions

In a distributed database environment, a given transaction updates data in multiple physical databases (protected by two-phase commit to ensure all nodes or none commit). In such an environment, all servers (whether Oracle or non-Oracle) that participate in a *serializable* transaction are required to support serializable isolation mode.

If a serializable transaction tries to update data in a database managed by a server that does not support serializable transactions, the transaction receives an error. The transaction can roll back and retry only when the remote server does support serializable transactions.

In contrast, *read committed* transactions can perform distributed transactions with servers that do not support serializable transactions.

Choosing an Isolation Level

Application designers and developers should choose an isolation level based on application performance and consistency needs as well as application coding requirements.

For environments with many concurrent users rapidly submitting transactions, designers must assess transaction performance requirements in terms of the expected transaction arrival rate and response time demands. Frequently, for high-performance environments, the choice of isolation levels involves a trade-off between consistency and concurrency (transaction throughput).

Application logic that checks database consistency must take into account the fact that reads do not block writes in either mode.

Both Oracle isolation modes provide high levels of consistency and concurrency (and performance) through the combination of row-level locking and Oracle's multiversion concurrency control system. Readers and writers don't block one another in Oracle; therefore, while queries still see consistent data, both read committed and serializable isolation provide a high level of concurrency for high performance, without the need for reading uncommitted ("dirty") data.

Choosing Read Committed Isolation

For many applications, read committed is the most appropriate isolation level. This is the isolation level used by applications running on Oracle releases previous to Release 7.3.

Read committed isolation can provide considerably more concurrency with a somewhat increased risk of inconsistent results (due to phantoms and non-repeatable reads) for some transactions.

Many high-performance environments with high transaction arrival rates require more throughput and faster response times than can be achieved with serializable isolation. Other environments that supports fe users with a very low transaction arrival rate also face very low risk of incorrect results due to phantoms and nonrepeatable reads. Read committed isolation is suitable for both of these environments.

Oracle read committed isolation provides transaction set consistency for every query (that is, every query sees data in a consistent state). Therefore, read committed isolation will suffice for many applications that might require a higher degree of isolation if run on other database management systems that do not use multiversion concurrency control.

Read committed isolation mode does not require application logic to trap the "Cannot serialize access" error and loop back to restart a transaction. In most applica-

tions, few transactions have a functional need to reissue the same query twice, so for many applications protection against phantoms and non-repeatable reads is not important. Therefore many developers choose read committed to avoid the need to write such error checking and retry code in each transaction.

Choosing Serializable Isolation

Oracle's serializable isolation is suitable for environments where there is relatively low chance that two concurrent transactions will modify the same rows and the relatively long-running transactions are primarily read-only. It is most suitable for environments with large databases and short transactions that update only a few rows.

Serializable isolation mode provides somewhat more consistency by protecting against phantoms and nonrepeatable reads and may be important where a read/write transaction executes a query more than once.

Unlike other implementations of serializable isolation, which lock blocks for read as well as write, Oracle provides nonblocking queries and the fine granularity of row-level locking, both of which reduce write/write contention. For applications that experience mostly read/write contention, Oracle serializable isolation can provide significantly more throughput than other systems. Therefore, some applications might be suitable for serializable isolation on Oracle but not on other systems.

All queries in an Oracle serializable transaction see the database as of a single point in time, so this isolation level is suitable where multiple consistent queries must be issued in a read-write transaction. A report-writing application that generates summary data and stores it in the database might use serializable mode because it provides the consistency that a READ ONLY transaction provides, but also allows INSERT, UPDATE, and DELETE.

Note: Transactions containing DML statements with subqueries should use serializable isolation to guarantee consistent read.

Coding serializable transactions requires extra work by the application developer (to check for the "Cannot serialize access" error and to roll back and retry the transaction). Similar extra coding is needed in other database management systems to manage deadlocks. For adherence to corporate standards or for applications that are run on multiple database management systems, it may be necessary to design transactions for serializable mode. Transactions that check for serializability failures and retry can be used with Oracle read committed mode (which does not generate serializability errors).

Serializable mode is probably not the best choice in an environment with relatively long transactions that must update the same rows accessed by a high volume of short update transactions. Because a longer running transaction is unlikely to be the first to modify a given row, it will repeatedly need to roll back, wasting work. (Note that a conventional read-locking "pessimistic" implementation of serializable mode would not be suitable for this environment either, because long-running transactions — even read transactions — would block the progress of short update transactions and vice versa.)

Application developers should take into account the cost of rolling back and retrying transactions when using serializable mode. As with read-locking systems, where deadlocks occur frequently, use of serializable mode requires rolling back the work done by aborted transactions and retrying them. In a high contention environment, this activity can use significant resources.

In most environments, a transaction that restarts after receiving the "Cannot serialize access" error is unlikely to encounter a second conflict with another transaction. For this reason it can help to execute those statements most likely to contend with other transactions as early as possible in a serializable transaction. However, there is no guarantee that the transaction will complete successfully, so the application should be coded to limit the number of retries.

Although Oracle serializable mode is compatible with SQL92 and offers many benefits compared with read-locking implementations, it does not provide semantics identical to such systems. Application designers must take into account the fact that reads in Oracle do not block writes as they do in other systems. Transactions that check for database consistency at the application level may require coding techniques such as the use of SELECT FOR UPDATE. This issue should be considered when applications using serializable mode are ported to Oracle from other environments.

How Oracle Locks Data

Locks are mechanisms that prevent destructive interaction between transactions accessing the same *resource* — either user objects (such as tables and rows) or system objects not visible to users (such as shared data structures in memory and data dictionary rows).

In all cases, Oracle automatically obtains necessary locks when executing SQL statements, so users need not be concerned with such details. Oracle automatically uses the lowest applicable level of restrictiveness to provide the highest degree of data concurrency yet also provide fail-safe data integrity. Oracle also allows the user to lock data manually.

For a complete description of the internal locks used by Oracle, see "Types of Locks" on page 23-18.

Transactions and Data Concurrency

Oracle provides data concurrency and integrity between transactions using its locking mechanisms. Because the locking mechanisms of Oracle are tied closely to transaction control, application designers need only define transactions properly, and Oracle automatically manages locking.

Keep in mind that Oracle locking is fully automatic and requires no user action. Implicit locking occurs for all SQL statements so that database users never need to lock any resource explicitly. Oracle's default locking mechanisms lock data at the lowest level of restrictiveness to guarantee data integrity while allowing the highest degree of data concurrency.

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 23-29.

Locking Modes

Oracle uses two modes of locking in a multiuser database:

exclusive lock mode	Prevents the associates resource from being shared. This lock mode is obtained to modify data. The first transaction to lock a resource exclusively is the only transaction that can alter the resource until the exclusive lock is released.
share lock mode	Allows the associated resource to be shared, depending on the operations involved. Multiple users reading data can share the data, holding share locks to prevent concurrent access by a writer (who needs an exclusive lock). Several transactions can acquire share locks on the same resource.

Lock Duration

All locks acquired by statements within a transaction are held for the duration of the transaction, preventing destructive interference (including dirty reads, lost updates, and destructive DDL operations) from concurrent transactions. The changes made by the SQL statements of one transaction become visible only to other transactions that start *after* the first transaction is committed.

Oracle releases all locks acquired by the statements within a transaction when you either commit or roll back the transaction. Oracle also releases locks acquired after a savepoint when rolling back to the savepoint. However, only transactions not waiting for the previously locked resources can acquire locks on the now available resources. Waiting transactions will continue to wait until after the original transaction commits or rolls back completely.

Data Lock Conversion Versus Lock Escalation

A transaction holds exclusive row locks for all rows inserted, updated, or deleted within the transaction. Because row locks are acquired at the highest degree of restrictiveness, no lock conversion is required or performed.

Oracle automatically converts a table lock of lower restrictiveness to one of higher restrictiveness as appropriate. For example, assume that a transaction uses a SELECT statement with the FOR UPDATE clause to lock rows of a table. As a result, it acquires the exclusive row locks and a row share table lock for the table. If the transaction later updates one or more of the locked rows, the row share table lock is automatically converted to a row exclusive table lock. For more information about table locks, see "Table Locks (TM)" on page 23-20.

Lock escalation occurs when numerous locks are held at one level of granularity (for example, rows) and a database raises the locks to a higher level of granularity (for example, table). For example, if a single user locks many rows in a table, some database will automatically escalate the user's row locks to a single table. The number of locks is reduced, but the restrictiveness of what is being locked is increased.

Oracle never escalates locks. Lock escalation greatly increases the likelihood of deadlocks (described below). Imagine the situation where the system is trying to escalate locks on behalf of transaction T1 but cannot because of the locks held by transaction T2. A deadlock is created if transaction T2 also requires lock escalation of the same data before it can proceed.

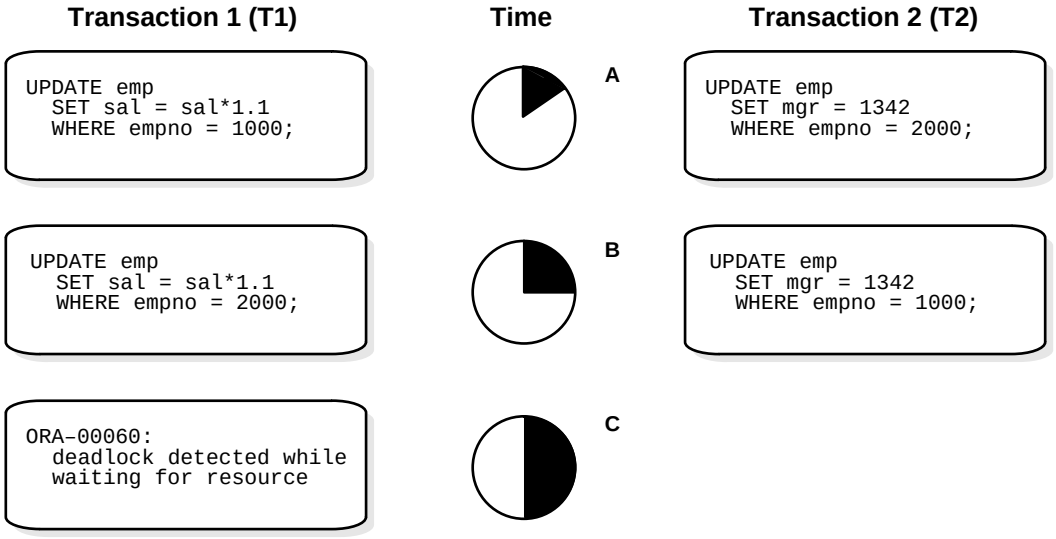
Deadlocks

A *deadlock* can occur when two or more users are waiting for data locked by each other. Deadlocks prevent some transactions from continuing to work. Figure 23–3 illustrates two transactions in a deadlock.

In Figure 23–3, no problem exists at time point A, as each transaction has a row lock on the row it attempts to update. Each transaction proceeds (without being terminated). However, each tries next to update the row currently held by the other transaction. Therefore, a deadlock results at time point B, because neither transac-

tion can obtain the resource it needs to proceed or terminate. It is a deadlock because no matter how long each transaction waits, the conflicting locks are held.

Figure 23–3 Two Transactions in a Deadlock



Deadlock Detection

Oracle automatically detects deadlock situations and resolves them by rolling back one of the statements involved in the deadlock, thereby releasing one set of the conflicting row locks. A corresponding message also is returned to the transaction that undergoes statement-level rollback. The statement rolled back is the one belonging to the transaction that detects the deadlock. Usually, the signalled transaction should be rolled back explicitly, but it can retry the rolled-back statement after waiting.

Note: In distributed transactions, local deadlocks are detected by analyzing a "waits for" graph, and global deadlocks are detected by a time-out. Once detected, nondistributed and distributed deadlocks are handled by the database and application in the same way.

Deadlocks most often occur when transactions explicitly override the default locking of Oracle. Because Oracle itself does no lock escalation and does not use read locks for queries, but does use row-level locking (rather than page-level locking), deadlocks occur infrequently in Oracle. See "Explicit (Manual) Data Locking" on page 23-29 for more information about manually acquiring locks and for an example of a deadlock situation.

Avoiding Deadlocks

Multitable deadlocks can usually be avoided if transactions accessing the same tables lock those tables in the same order, either through implicit or explicit locks. For example, all application developers might follow the rule that when both a master and detail table are updated, the master table is locked first and then the detail table. If such rules are properly designed and then followed in all applications, deadlocks are very unlikely to occur.

When you know you will require a sequence of locks for one transaction, you should consider acquiring the most exclusive (least compatible) lock first.

Types of Locks

Oracle automatically uses different types of locks to control concurrent access to data and to prevent destructive interaction between users. Oracle automatically locks a resource on behalf of a transaction to prevent other transactions from doing something also requiring exclusive access to the same resource. The lock is released automatically when some event occurs so that the transaction no longer requires the resource.

Throughout its operation, Oracle automatically acquires different types of locks at different levels of restrictiveness depending on the resource being locked and the operation being performed.

Oracle locks fall into one of the following general categories:

DML locks (data locks)	DML locks protect data. For example, table locks lock entire tables, row locks lock selected rows.
DDL locks (dictionary locks)	DDL locks protect the structure of schema objects — for example, the definitions of tables and views.
internal locks and latches	Internal locks and latches protect internal database structures such as datafiles. Internal locks and latches are entirely automatic.

distributed locks	Distributed locks ensure that the data and other resources distributed among the various instances of an Oracle Parallel Server remain consistent. Distributed locks are held by instances rather than transactions. They communicate the current status of a resource among the instances of an Oracle Parallel Server.
parallel cache management (PCM) locks	Parallel cache management locks are distributed locks that cover one or more data blocks (table or index blocks) in the buffer cache. PCM locks do not lock any rows on behalf of transactions.

This chapter discusses DML locks, DDL locks, and internal locks, respectively.

> **Additional Information:** See *Oracle8 Parallel Server Concepts and Administration* for more information about distributed locks and PCM locks.

DML (Data) Locks

The purpose of a DML (data) lock is to guarantee the integrity of data being accessed concurrently by multiple users. DML locks prevent destructive interference of simultaneous conflicting DML and/or DDL operations. For example, Oracle DML locks guarantee that a specific row in a table can be updated by only one transaction at a time and that a table cannot be dropped if an uncommitted transaction contains an insert into the table.

DML operations can acquire data locks at two different levels: for specific rows and for entire tables. The following sections explain row and table locks.

> **Note:** The acronym in parentheses after each type of lock or lock mode in the following sections is the abbreviation used in the Locks Monitor of Oracle Enterprise Manager. Oracle Enterprise Manager might display TM for any table lock, rather than indicate the mode of table lock (such as RS or SRX).

Row Locks (TX)

The only DML locks Oracle acquires automatically are row-level locks. There is no limit to the number of row locks held by a statement or transaction, and Oracle does not escalate locks from the row level to a coarser granularity. Row locking pro-

vides the finest grain locking possible and so provides the best possible concurrency and throughput.

The combination of multiversion concurrency control and row-level locking means that users contend for data only when accessing the same rows, specifically:

- Readers of data do not wait for writers of the same data rows.
- Writers of data do not wait for readers of the same data rows (unless SELECT... FOR UPDATE is used, which specifically requests a lock for the reader).
- Writers only wait for other writers if they attempt to update the same rows at the same time.

> **Note:** Readers of data may have to wait for writers of the same data blocks in some very special cases of pending distributed transactions.

A transaction acquires an exclusive DML lock for each individual row modified by one of the following statements: INSERT, UPDATE, DELETE, and SELECT with the FOR UPDATE clause.

A modified row is **always** locked exclusively so that other users cannot modify the row until the transaction holding the lock is committed or rolled back. Row locks are always acquired automatically by Oracle as a result of the statements listed above.

If a transaction obtains a row lock for a row, the transaction also acquires a table lock for the corresponding table. A table lock is also necessary 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 23-26 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 23-27.)

A table lock can be held in any of several modes: row share (RS), row exclusive (RX), share (S), share row exclusive (SRX), and exclusive (X). The restrictiveness of a table lock’s mode determines the modes in which other table locks on the same table can be obtained and held.

Table 23–2 shows the table lock modes that statements acquire and operations that those locks permit and prohibit.

Table 23–2 Summary of Table Locks

SQL Statement	Mode of Table Lock	Lock Modes Permitted?				
		RS	RX	S	SRX	X
SELECT...FROM table...	none	Y	Y	Y	Y	Y
INSERT INTO table ...	RX	Y	Y	N	N	N
UPDATE table ...	RX	Y*	Y*	N	N	N
DELETE FROM table ...	RX	Y*	Y*	N	N	N
SELECT ... FROM table FOR UPDATE OF ...	RS	Y*	Y*	Y*	Y*	N
LOCK TABLE table IN ROW SHARE MODE	RS	Y	Y	Y	Y	N
LOCK TABLE table IN 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
	RS: row share RX: row exclusive S: share SRX: share row exclusive X: exclusive	*Yes, if no conflicting row locks are held by another transaction; otherwise, waits occur.				

The following sections explain each mode of table lock, from least restrictive to most restrictive. Each section describes the mode of table lock, the actions that cause the transaction to acquire a table lock in that mode, and which actions are permitted and prohibited in other transactions by a lock in that mode. For more information about manual locking, see "Explicit (Manual) Data Locking" on page 23-29.

Row Share Table Locks (RS) A row share table lock (also sometimes called a *subshare table lock, SS*) indicates that the transaction holding the lock on the table has locked rows in the table and intends to update them. A row share table lock is automatically acquired for a *table* when one of the following SQL statements is executed:

```
SELECT . . . FROM table . . . FOR UPDATE OF . . . ;

LOCK TABLE table IN ROW SHARE MODE;
```

A row share table lock is the least restrictive mode of table lock, offering the highest degree of concurrency for a table.

Permitted Operations: A row share table lock held by a transaction allows other transactions to query, insert, update, delete, or lock rows concurrently in the same table. Therefore, other transactions can obtain simultaneous row share, row exclusive, share, and share row exclusive table locks for the same table.

Prohibited Operations: A row share table lock held by a transaction prevents other transactions from exclusive write access to the same table using only the following statement:

```
LOCK TABLE table IN EXCLUSIVE MODE;
```

Row Exclusive Table Locks (RX) A row exclusive table lock (also called a *subexclusive table lock, SX*) generally indicates that the transaction holding the lock has made one or more updates to rows in the table. A row exclusive table lock is acquired automatically for a *table* modified by the following types of statements:

```
INSERT INTO table . . . ;

UPDATE table . . . ;

DELETE FROM table . . . ;

LOCK TABLE table IN ROW EXCLUSIVE MODE;
```

A row exclusive table lock is slightly more restrictive than a row share table lock.

Permitted Operations: A row exclusive table lock held by a transaction allows other transactions to query, insert, update, delete, or lock rows concurrently in the same table. Therefore, row exclusive table locks allow multiple transactions to obtain simultaneous row exclusive and row share table locks for the same table.

Prohibited Operations: A row exclusive table lock held by a transaction prevents other transactions from manually locking the table for exclusive reading or writing. Therefore, other transactions cannot concurrently lock the table using the following statements:

```
LOCK TABLE table IN SHARE MODE;

LOCK TABLE table IN SHARE EXCLUSIVE MODE;

LOCK TABLE table IN EXCLUSIVE MODE;
```

Share Table Locks (S) A share table lock is acquired automatically for the *table* specified in the following statement:

```
LOCK TABLE table IN SHARE MODE;
```

Permitted Operations: A share table lock held by a transaction allows other transactions only to query the table, to lock specific rows with SELECT . . . FOR UPDATE, or to execute LOCK TABLE . . . IN SHARE MODE statements successfully; no updates are allowed by other transactions. Multiple transactions can hold share table locks for the same table concurrently. In this case, no transaction can update the table (even if a transaction holds row locks as the result of a SELECT statement with the FOR UPDATE clause). Therefore, a transaction that has a share table lock can update the table only if no other transactions also have a share table lock on the same table.

Prohibited Operations: A share table lock held by a transaction prevents other transactions from modifying the same table and from executing the following statements:

```
LOCK TABLE table IN SHARE ROW EXCLUSIVE MODE;

LOCK TABLE table IN EXCLUSIVE MODE;

LOCK TABLE table IN ROW EXCLUSIVE MODE;
```

Share Row Exclusive Table Locks (SRX) A share row exclusive table lock (also sometimes called a *share-subexclusive table lock, SSX*) is more restrictive than a share table lock. A share row exclusive table lock is acquired for a *table* as follows:

```
LOCK TABLE table IN SHARE ROW EXCLUSIVE MODE;
```

Permitted Operations: Only one transaction at a time can acquire a share row exclusive table lock on a given table. A share row exclusive table lock held by a transaction allows other transactions to query or lock specific rows using SELECT with the FOR UPDATE clause, but not to update the table.

Prohibited Operations: A share row exclusive table lock held by a transaction prevents other transactions from obtaining row exclusive table locks and modifying the same table. A share row exclusive table lock also prohibits other transactions from obtaining share, share row exclusive, and exclusive table locks, which prevents other transactions from executing the following statements:

```
LOCK TABLE table IN SHARE MODE;

LOCK TABLE table IN SHARE ROW EXCLUSIVE MODE;

LOCK TABLE table IN ROW EXCLUSIVE MODE;

LOCK TABLE table IN EXCLUSIVE MODE;
```

Exclusive Table Locks (X) An exclusive table lock is the most restrictive mode of table lock, allowing the transaction that holds the lock exclusive write access to the table. An exclusive table lock is acquired for a *table* as follows:

```
LOCK TABLE table IN EXCLUSIVE MODE;
```

Permitted Operations: Only one transaction can obtain an exclusive table lock for a table. An exclusive table lock permits other transactions only to query the table.

Prohibited Operations: An exclusive table lock held by a transaction prohibits other transactions from performing any type of DML statement or placing any type of lock on the table.

DML Locks Automatically Acquired for DML Statements

The previous sections explained the different types of data locks, the modes in which they can be held, when they can be obtained, when they are obtained, and what they prohibit. The following sections summarize how Oracle automatically locks data on behalf of different DML operations.

Table 23–3 summarizes the information in the following sections.

Table 23–3 Locks Obtained By DML Statements

DML Statement	Row Locks?	Mode of Table Lock
SELECT ... FROM table		
INSERT INTO table ...	X	RX
UPDATE table ...	X	RX
DELETE FROM table ...	X	RX
SELECT ... FROM table ... FOR UPDATE OF ...	X	RS
LOCK TABLE table IN ...		
ROW SHARE MODE		RS
ROW EXCLUSIVE MODE		RX
SHARE MODE		S
SHARE EXCLUSIVE MODE		SRX
EXCLUSIVE MODE		X
	X: exclusive RX: row exclusive	RS: row share S: share SRX: share row exclusive

Default Locking for Queries Queries are the SQL statements least likely to interfere with other SQL statements because they only read data. INSERT, UPDATE, and DELETE statements can have implicit queries as part of the statement. Queries include the following kinds of statements:

```
SELECT

INSERT . . . SELECT . . . ;

UPDATE . . . ;

DELETE . . . ;
```

They do **not** include the following statement:

```
SELECT . . . FOR UPDATE OF . . . ;
```

The following characteristics are true of all queries that do not use the FOR UPDATE clause:

- A query acquires no data locks. Therefore, other transactions can query and update a table being queried, including the specific rows being queried. Because queries lacking FOR UPDATE clauses do not acquire any data locks to block other operations, such queries are often referred to in Oracle as *nonblocking queries*.
- A query does not have to wait for any data locks to be released; it can always proceed. (Queries may have to wait for data locks in some very specific cases of pending distributed transactions.)

Default Locking for INSERT, UPDATE, DELETE, and SELECT ... FOR UPDATE The locking characteristics of INSERT, UPDATE, DELETE, and SELECT ... FOR UPDATE statements are as follows:

- The transaction that contains a DML statement acquires exclusive row locks on the rows modified by the statement. Other transactions cannot update or delete the locked rows until the locking transaction either commits or rolls back.
- The transaction that contains a DML statement does not need to acquire row locks on any rows selected by a subquery or an implicit query, such as a query in a WHERE clause. A subquery or implicit query in a DML statement is guaranteed to be consistent as of the start of the query and does not see the effects of the DML statement it is part of.
- A query in a transaction can see the changes made by previous DML statements in the same transaction, but cannot see the changes of other transactions begun after its own transaction.
- In addition to the necessary exclusive row locks, a transaction that contains a DML statement acquires at least a row exclusive table lock on the table that contains the affected rows. If the containing transaction already holds a share, share row exclusive, or exclusive table lock for that table, the row exclusive table lock is not acquired. If the containing transaction already holds a row share table lock, Oracle automatically converts this lock to a row exclusive table lock.

DDL Locks (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 schema objects referenced in the procedure definition. The DDL locks prevent objects referenced in the procedure from being altered or dropped before the procedure compilation is complete.

Oracle acquires a dictionary lock automatically on behalf of any DDL transaction requiring it. Users cannot explicitly request DDL locks. Only individual schema objects that are modified or referenced are locked during DDL operations; the whole data dictionary is never locked.

DDL locks fall into three categories: exclusive DDL locks, share DDL locks, and breakable parse locks.

Exclusive DDL Locks

Most DDL operations (except for those listed in the next section, "Share DDL Locks") require exclusive DDL locks for a resource to prevent destructive interference with other DDL operations that might modify or reference the same schema object. For example, a DROP TABLE operation is not allowed to drop a table while an ALTER TABLE operation is adding a column to it, and vice versa.

During the acquisition of an exclusive DDL lock, if another DDL lock is already held on the schema object by another operation, the acquisition waits until the older DDL lock is released and then proceeds.

DDL operations also acquire DML locks (data locks) on the schema object to be modified.

Share DDL Locks

Some DDL operations require share DDL locks for a resource to prevent destructive interference with conflicting DDL operations, but allow data concurrency for similar DDL operations. For example, when a CREATE PROCEDURE statement is executed, the containing transaction acquires share DDL locks for all referenced tables. Other transactions can concurrently create procedures that reference the same tables and therefore acquire concurrent share DDL locks on the same tables, but no transaction can acquire an exclusive DDL lock on any referenced table. No transaction can alter or drop a referenced table. As a result, a transaction that holds a share DDL lock is guaranteed that the definition of the referenced schema object will remain constant for the duration of the transaction.

A share DDL lock is acquired on a schema object for DDL statements that include the following 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 schema object it references. Parse locks are acquired so that the associated shared SQL area can be invalidated if a referenced object is altered or dropped. See Chapter 19, "Oracle Dependency Management", 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 for that statement remains in the shared pool.

Duration of DDL Locks

The duration of a DDL lock depends on its type. Exclusive and share DDL locks last for the duration of DDL statement execution and automatic commit. A parse lock persists as long as the associated SQL statement remains in the shared pool.

DDL Locks and Clusters

A DDL operation on a cluster acquires exclusive DDL locks on the cluster and on all tables and snapshots in the cluster. A DDL operation on a table or snapshot in a cluster acquires a share lock on the cluster, in addition to a share or exclusive DDL lock on the table or snapshot. The share DDL lock on the cluster prevents another operation from dropping the cluster while the first operation proceeds.

Latches and Internal Locks

Latches and internal locks protect internal database and memory structures. Both are inaccessible to users, because users have no need to control over their occurrence or duration. The following information will help you interpret the Oracle Enterprise Manager or 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

Internal locks are higher-level, more complex mechanisms than latches and serve a variety of purposes.

Dictionary Cache Locks These locks are of very short duration and are held on entries in dictionary caches while the entries are being modified or used. They guarantee that statements being parsed do not see inconsistent object definitions.

Dictionary cache locks can be shared or exclusive. Shared locks are released when the parse is complete. Exclusive locks are released when the DDL operation is complete.

File and Log Management Locks These locks protect various files. For example, one lock protects the control file so that only one process at a time can change it. Another lock coordinates the use and archiving of the redo log files. Datafiles are locked to ensure that multiple instances mount a database in shared mode or that one instance mounts it in exclusive mode. Because file and log locks indicate the status of files, these locks are necessarily held for a long time.

File and log locks are of particular importance if you are using the Oracle Parallel Server.

> **Additional Information:** See *Oracle8 Parallel Server Concepts and Administration* for more information about locks.

Tablespace and Rollback Segment Locks These locks protect tablespaces and rollback segments. For example, all instances accessing a database must agree on whether a tablespace is online or offline. Rollback segments are locked so that only one instance can write to a segment.

Explicit (Manual) Data Locking

Oracle always performs locking automatically to ensure data concurrency, data integrity, and statement-level read consistency. However, you can override the Oracle default locking mechanisms. Overriding the default locking is useful in situations such as these:

- Applications require transaction-level read consistency or “repeatable reads”. In other words, queries in them must produce consistent data for the duration of the transaction, not reflecting changes by other transactions. You can achieve transaction-level read consistency by using explicit locking, read-only transactions, serializable transactions, or by overriding default locking.

- Applications require that a transaction have exclusive access to a resource so that the transaction does not have to wait for other transactions to complete.

Oracle's automatic locking can be overridden at two levels:

transaction — Transactions that include the following SQL statements override Oracle's default locking:

- the SET TRANSACTION ISOLATION LEVEL command
- the LOCK TABLE command (which locks either a table or, when used with views, the underlying base tables)
- the SELECT... FOR UPDATE command

Locks acquired by these statements are released after the transaction commits or rolls back.

session — A session can set the required transaction isolation level with the ALTER SESSION command.

Note: If Oracle's default locking is overridden at any level, the database administrator or application developer should ensure that the overriding locking procedures operate correctly. The locking procedures must satisfy the following criteria: data integrity is guaranteed, data concurrency is acceptable, and deadlocks are not possible or are appropriately handled.

Additional Information: See the *Oracle8 SQL Reference* for detailed descriptions of the SQL statements LOCK TABLE and SELECT ... FOR UPDATE.

Examples of Concurrency under Explicit Locking

The following illustration shows how Oracle maintains data concurrency, integrity, and consistency when LOCK TABLE and SELECT with the FOR UPDATE clause statements are used.

Note: For brevity, the message text for ORA-00054 ("resource busy and acquire with NOWAIT specified") is not included. User-entered text is in **bold**.

Transaction 1	Time Point	Transaction 2
LOCK TABLE scott.dept	1	
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	5	
SET loc = 'NEW YORK'		
WHERE deptno = 20;		
(waits because T2 has locked same rows)		
	6	**ROLLBACK;**
		(releases row locks)

Transaction 1	Time Point	Transaction 2
1 row processed.	7	
ROLLBACK;		
LOCK TABLE scott.dept	8	
IN ROW EXCLUSIVE MODE;		
Statement processed.		
	9	**LOCK TABLE scott.dept**
		IN EXCLUSIVE MODE
		NOWAIT;
		ORA-00054
	10	**LOCK TABLE scott.dept**
		IN SHARE ROW EXCLUSIVE
		MODE NOWAIT;
		ORA-00054
	11	**LOCK TABLE scott.dept**
		IN SHARE ROW EXCLUSIVE
		MODE NOWAIT;
		ORA-00054
	12	**UPDATE scott.dept**
		SET loc = 'NEW YORK'
		WHERE deptno = 20;
		1 row processed.
	13	**ROLLBACK;**

Transaction 1	Time Point	Transaction 2
SELECT loc **FROM scott.dept** **WHERE deptno = 20** **FOR UPDATE OF loc;** LOC - - - - - - DALLAS 1 row selected.	14	
	15	**UPDATE scott.dept** **SET loc = 'NEW YORK'** **WHERE deptno = 20;** *(waits because T1 has locked same rows)*
ROLLBACK;	16	
	17	1 row processed. *(conflicting locks were released)* **ROLLBACK;**
LOCK TABLE scott.dept **IN ROW SHARE MODE** Statement processed	18	
	19	**LOCK TABLE scott.dept** **IN EXCLUSIVE MODE NOWAIT;** ORA-00054
	20	**LOCK TABLE scott.dept** **IN SHARE ROW EXCLUSIVE** **MODE NOWAIT;** ORA-00054

Transaction 1	Time Point	Transaction 2
	21	**LOCK TABLE scott.dept**
		IN SHARE MODE;
		Statement processed.
	22	**SELECT loc**
		FROM scott.dept
		WHERE deptno = 20;
		LOC
		- - - - - -
		DALLAS
		1 row selected.
	23	**SELECT loc**
		FROM scott.dept
		WHERE deptno = 20
		FOR UPDATE OF loc;
		LOC
		- - - - - -
		DALLAS
		1 row selected.
	24	**UPDATE scott.dept**
		SET loc = 'NEW YORK'
		WHERE deptno = 20;
		(waits because T1 holds conflicting table lock)
ROLLBACK;	25	
	26	1 row processed.
		(conflicting table lock released)
		ROLLBACK;

Transaction 1	Time Point	Transaction 2
LOCK TABLE scott.dept **IN SHARE ROW** **EXCLUSIVE MODE;**	27	
Statement processed.		
	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

Transaction 1	Time Point	Transaction 2
	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)*
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.

Transaction 1	Time Point	Transaction 2
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.

Transaction 1	Time Point	Transaction 2
SELECT loc	53	
FROM scott.dept		
WHERE deptno = 10;		
LOC		
- - - - - -		
BOSTON		
(T1 does not see uncommitted data)		
	54	**COMMIT;**
SELECT loc	55	
FROM scott.dept		
WHERE deptno = 10;		
LOC		
- - - - - -		
(same results seen even after T2 commits)		
COMMIT;	56	
SELECT loc	57	
FROM scott.dept		
WHERE deptno = 10;		
LOC		
- - - - - -		
NEW YORK		
(committed data is seen)		

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.

> **Additional Information:** See the *Oracle8 Application Developer's Guide* for more information about Oracle Lock Management services.

24

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

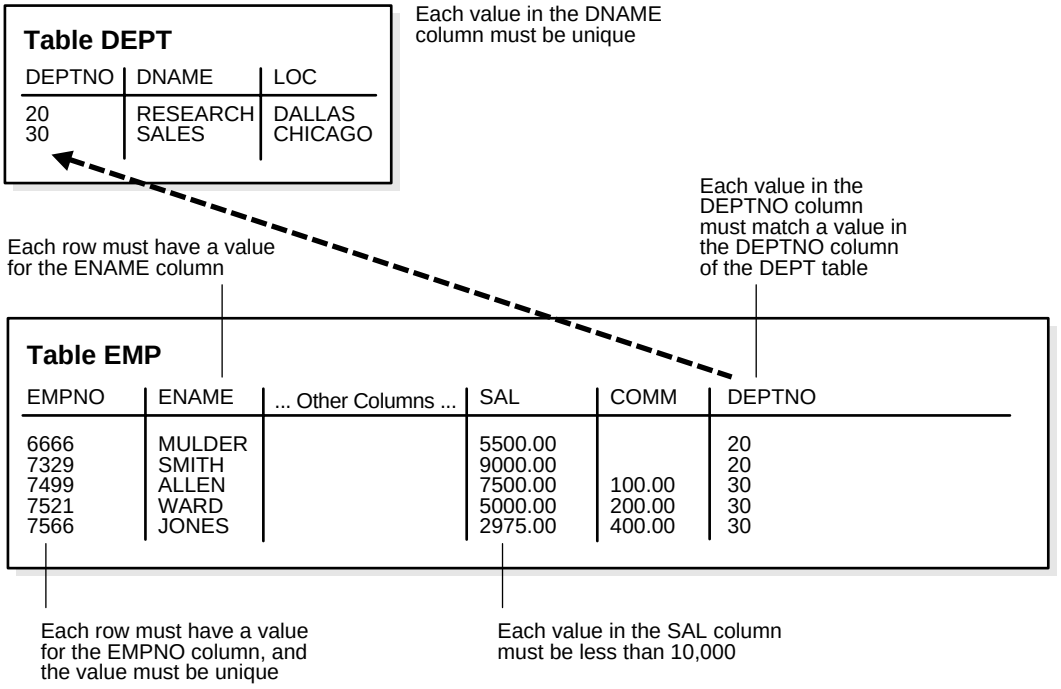
- Definition of Data Integrity
- An Introduction to Integrity Constraints
- Types of Integrity Constraints
- The Mechanisms of Constraint Checking
- Deferred Constraint Checking
- Enabled, Disabled, and Enable Novalidate Constraints

> **Additional Information:** If you are using Trusted Oracle, see your *Trusted Oracle* documentation 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 24–1.

Figure 24–1 Examples of Data Integrity



Note that some columns in each table have specific rules that constrain the data contained within them.

Types of Data Integrity

This section describes the rules that can be applied to table columns to enforce different types of data integrity.

Nulls

A null is a rule defined on a single column that allows or disallows inserts or updates of rows containing a null (the absence of a value) in that column.

Unique Column Values

A unique value defined on a column (or set of columns) allows the insert or update of a row only if it contains a unique value in that column (or set of columns).

Primary Key Values

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

Referential Integrity

A rule defined on a key (a column or set of columns) in one table that guarantees that the values in that key match the values in a key in a related table (the referenced value).

Referential integrity also includes the rules that dictate what types of data manipulation are allowed on referenced values and how these actions affect dependent values. The rules associated with referential integrity are:

Restrict	Disallows the update or deletion of referenced data.
Set to Null	When referenced data is updated or deleted, all associated dependent data is set to NULL.
Set to Default	When referenced data is updated or deleted, all associated dependent data is set to a default value.
Cascade	When referenced data is updated, all associated dependent data is correspondingly updated; when a referenced row is deleted, all associated dependent rows are deleted.
No Action	Disallows the update or deletion of referenced data. This differs from RESTRICT in that it is checked at the end of the statement, or at the end of the transaction if the constraint is deferred. (Oracle uses No Action as its default action.)

Complex Integrity Checking

Complex integrity checking is a user-defined rule for a column (or set of columns) that allows or disallows inserts, updates, or deletes of a row based on the value it contains for the column (or set of columns).

How Oracle Enforces Data Integrity

Oracle enables you to define and enforce each type of data integrity rule defined in the previous section. Most of these rules are easily defined using integrity constraints or database triggers.

Integrity Constraints

An integrity constraint is a declarative method of defining a rule for a column of a table. Oracle supports the following integrity constraints:

- NOT NULL constraints for the rules associated with nulls in a column
- UNIQUE key constraints for the rule associated with unique column values
- PRIMARY KEY constraints for the rule associated with primary identification values
- FOREIGN KEY constraints for the rules associated with referential integrity. Oracle currently supports the use of FOREIGN KEY integrity constraints to define the referential integrity actions, including
  - update and delete No Action
  - delete CASCADE
- CHECK constraints for complex integrity rules

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

Database Triggers

Oracle also allows you to enforce integrity rules with a nondeclarative approach using database triggers (stored database procedures automatically invoked on insert, update, or delete operations). For more information and examples of database triggers used to enforce data integrity, see Chapter 18, “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 you want to associate 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 information error message.

The integrity constraints implemented in Oracle fully comply with ANSI X3.135-1989 and ISO 9075-1989 standards.

Advantages of Integrity Constraints

This section describes some of the advantages that integrity constraints have over other alternatives, which include:

- enforcing business rules in the code of a database application
- using stored procedures to completely control access to data
- enforcing business rules with triggered stored database procedures (see Chapter 18, “Database Triggers”)

Declarative Ease

You define integrity constraints using SQL commands. When you define or alter a table, no additional programming is required. The SQL statements 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 the stored procedure solution to data integrity controls data access, but integrity constraints do not eliminate the flexibility of ad hoc data access.

Centralized Rules

Integrity constraints are defined for tables (not an application) and are stored in the data dictionary. Any data entered by any application must adhere to the same integrity constraints associated with the table. By moving business rules from application code to centralized integrity constraints, the tables of a database are guaranteed to contain valid data, no matter which database application manipulates the information. Stored procedures cannot provide the same advantage of centralized rules stored with a table. Database triggers can provide this benefit, but the complexity of implementation is far greater than the declarative approach used for integrity constraints.

Maximum Application Development Productivity

If a business rule enforced by an integrity constraint changes, the administrator need only change that integrity constraint and all applications automatically adhere to the modified constraint. In contrast, if the business rule were enforced by the code of each database application, developers would have to modify all application source code and recompile, debug, and test the modified applications.

Immediate User Feedback

Oracle stores specific information about each integrity constraint in the data dictionary. You can design database applications to use this information to provide immediate user feedback about integrity constraint violations, even before Oracle executes and checks the SQL statement. For example, a SQL*Forms application can use integrity constraint definitions stored in the data dictionary to check for violations as values are entered into the fields of a form, even before the application issues a statement.

Superior Performance

The semantics of integrity constraint declarations are clearly defined, and performance optimizations are implemented for each specific declarative rule. The Oracle query optimizer can use declarations to learn more about data to improve overall query performance. (Also, taking integrity rules out of application code and database triggers guarantees that checks are only made when necessary.)

Flexibility for Data Loads and Identification of Integrity Violations

You can disable integrity constraints temporarily so that large amounts of data can be loaded without the overhead of constraint checking. When the data load is complete, you can easily enable the integrity constraints, and you can automatically report any new rows that violate integrity constraints to a separate exceptions table.

The Performance Cost of Integrity Constraints

The advantages of enforcing data integrity rules do not come without some loss in performance. In general, the "cost" of including an integrity constraint is, at most, the same as executing a SQL statement that evaluates the constraint.

Types of Integrity Constraints

You can use the following integrity constraints to impose restrictions on the input of column values:

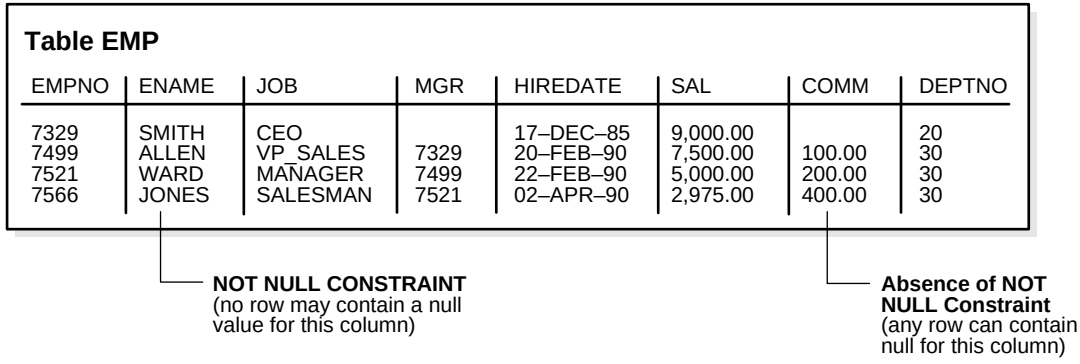
- NOT NULL Integrity Constraints
- UNIQUE Key Integrity Constraints
- PRIMARY KEY Integrity Constraints
- FOREIGN KEY (Referential) Integrity Constraints
- CHECK Integrity Constraints

NOT NULL Integrity Constraints

By default, all columns in a table allow nulls (the absence of a value). A NOT NULL constraint requires a column of a table contain no null values. For example, you can define a NOT NULL constraint to require that a value be input in the ENAME column for every row of the EMP table.

Figure 24–2 illustrates a NOT NULL integrity constraint.

Figure 24–2 NOT NULL Integrity Constraints

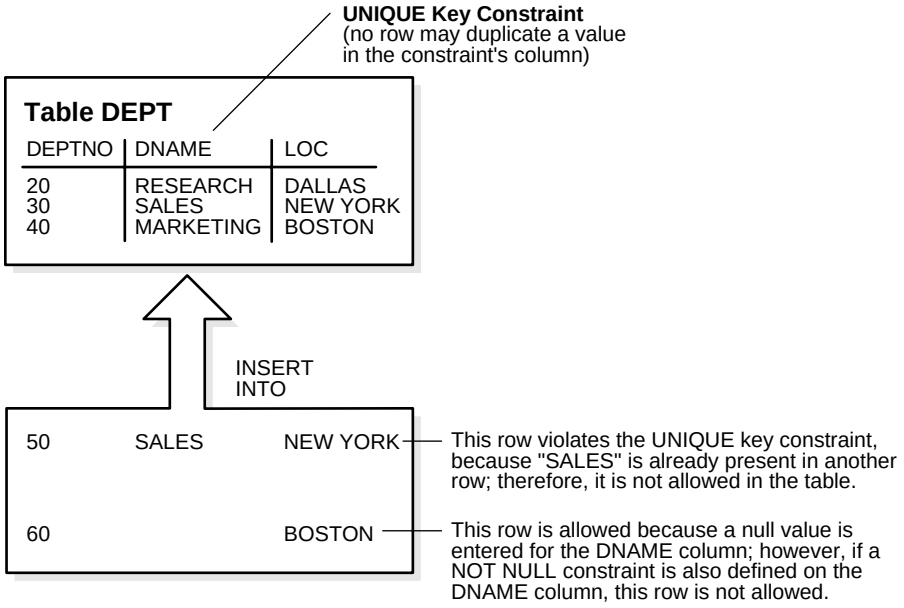


UNIQUE Key Integrity Constraints

A UNIQUE key integrity constraint requires that every value in a column or set of columns (key) be unique — that is, no two rows of a table have duplicate values in a specified column or set of columns.

For example, in Figure 24–3 a UNIQUE key constraint is defined on the DNAME column of the DEPT table to disallow rows with duplicate department names.

Figure 24–3 A UNIQUE Key Constraint



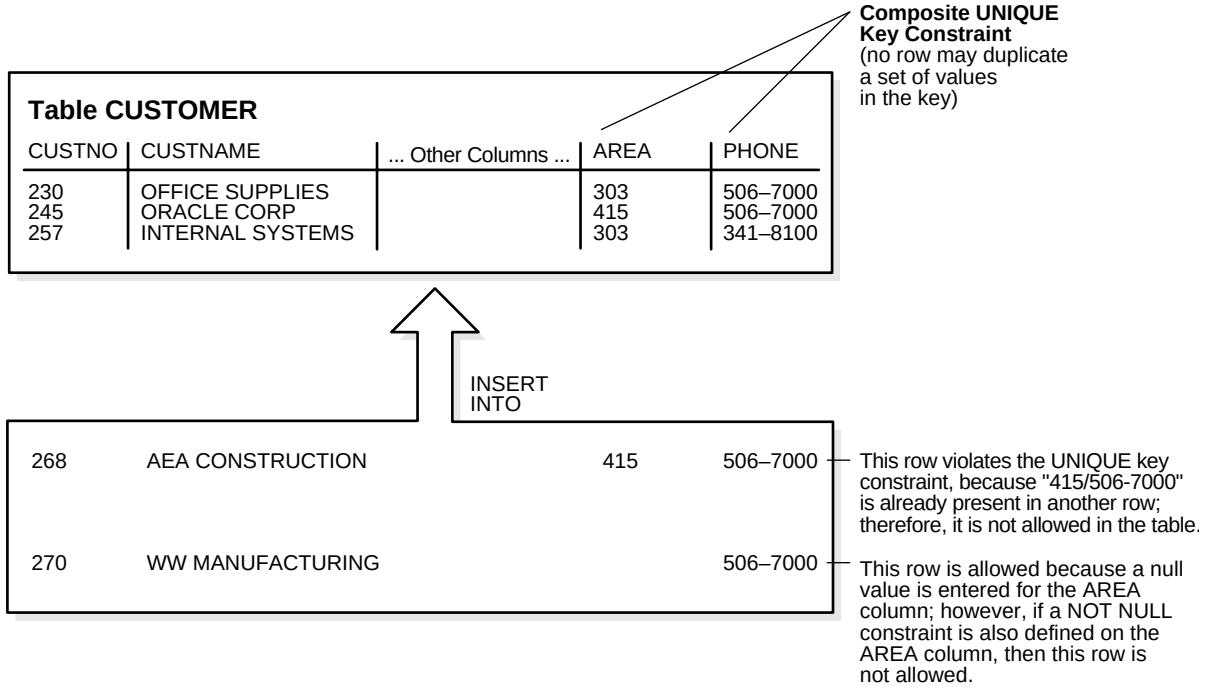
Unique Keys

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 column or set of columns used in the definition of the integrity constraint.

If the UNIQUE key consists of more than one column, that group of columns is said to be a *composite unique key*. For example, in Figure 24–4 the CUSTOMER table

has a UNIQUE key constraint defined on the composite unique key: the AREA and PHONE columns.

Figure 24–4 A Composite UNIQUE Key Constraint



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

UNIQUE Key Constraints and Indexes

Oracle enforces unique integrity constraints with indexes. (In Figure 24–4, Oracle enforces the UNIQUE key constraint by implicitly creating a unique index on the composite unique key.) Therefore, composite UNIQUE key constraints have the same limitations imposed on composite indexes: up to 32 columns can constitute a composite unique key, and the total size (in bytes) of a key value cannot exceed approximately half the associated database's block size. If a useable index exists

when a unique key constraint is created, the constraint will use that index rather than implicitly creating a new one.

Combining UNIQUE Key and NOT NULL Integrity Constraints

In Figure 24–3 and Figure 24–4, UNIQUE key constraints allow the input of nulls unless you also define NOT NULL constraints for the same columns. In fact, any number of rows can include nulls for columns without NOT NULL constraints because nulls are not considered equal to anything. A null in a column (or in all columns of a composite UNIQUE key) always satisfies a UNIQUE key constraint.

Columns with both unique keys and NOT NULL integrity constraints are common. This combination forces the user to enter values in the unique key and also eliminates the possibility that any new row's data will ever conflict with an existing row's data.

Note: Because of the search mechanism for UNIQUE constraints on more than one column, you cannot have identical values in the non-null columns of a partially null composite UNIQUE key constraint.

PRIMARY KEY Integrity Constraints

Each table in the database can have at most one PRIMARY KEY constraint. The values in the group of one or more columns subject to this constraint constitute the unique identifier of the row. In effect, each row is named by its primary key values.

The Oracle implementation of the PRIMARY KEY integrity constraint guarantees that both of the following are true:

- No two rows of a table have duplicate values in the specified column or set of columns.
- The primary key columns do not allow nulls (that is, a value must exist for the primary key columns in each row).

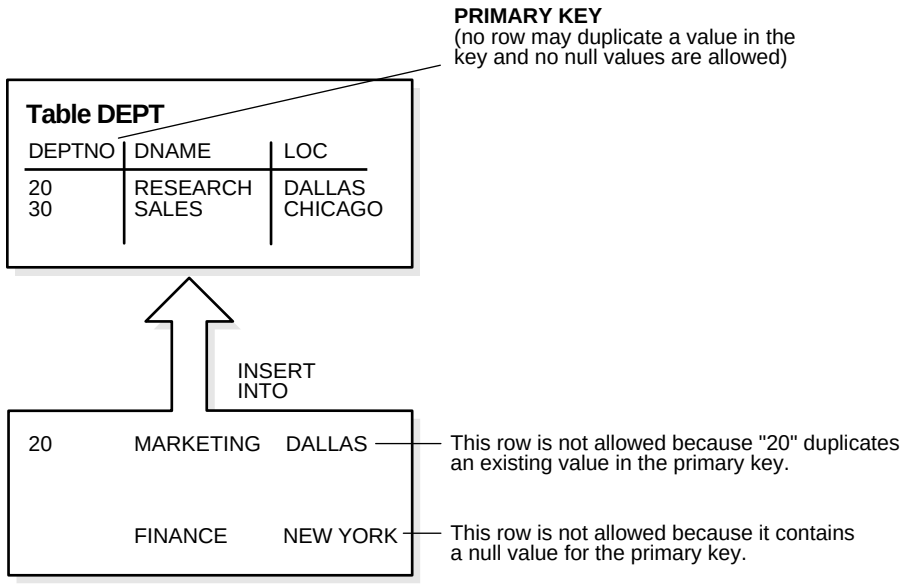
Primary Keys

The 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 24–5 illustrates a PRIMARY KEY constraint in the DEPT table and examples of rows that violate the constraint.

Figure 24–5 A Primary Key Constraint



PRIMARY KEY Constraints and Indexes

Oracle enforces all PRIMARY KEY constraints using indexes. In Figure 24–5, 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

Oracle enforces primary key constraints using indexes, and composite primary key constraints are limited to 32 columns, which is the same limitation imposed on composite indexes. The name of the index is the same as the name of the constraint. Also, you can specify the storage options for the index by including the ENABLE clause in the CREATE TABLE or ALTER TABLE statement used to create the constraint. If a useable index exists when a primary key constraint is created, the primary key constraint will use that index rather than implicitly creating a new one.

FOREIGN KEY (Referential) Integrity Constraints

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

Several terms are associated with referential integrity constraints:

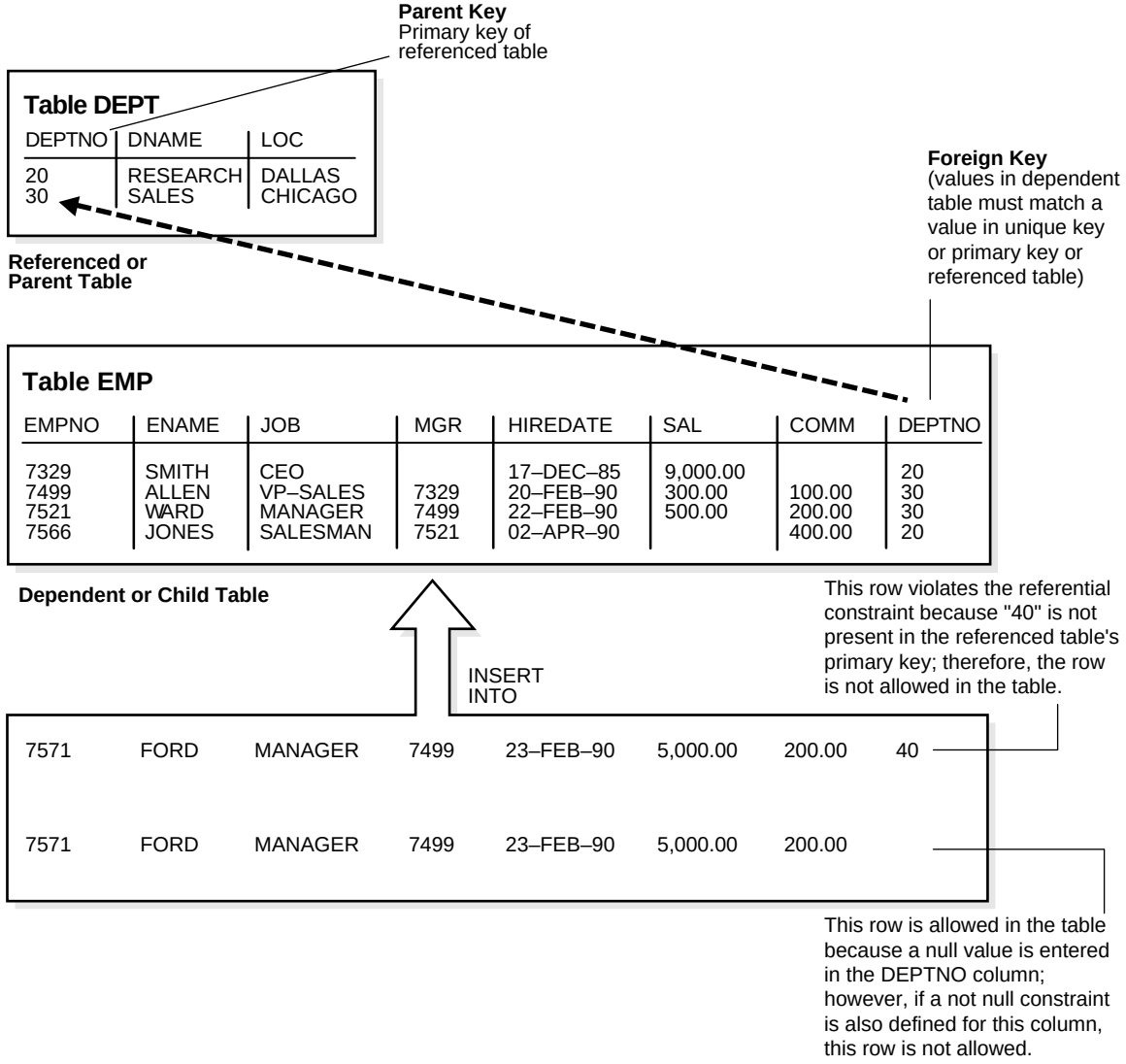
foreign key	The column or set of columns included in the definition of the referential integrity constraint that reference a referenced key (see the following).
referenced key	The unique key or primary key of the same or different table that is referenced by a foreign key.
dependent or child table	The table that includes the foreign key. Therefore, it is the table that is dependent on the values present in the referenced unique or primary key.
referenced or parent table	The table that is referenced by the child table's foreign key. It is this table's referenced key that determines whether specific inserts or updates are allowed in the child table.

A referential integrity constraint requires that for each row of a table, the value in the foreign key matches a value in a parent key.

Figure 24–6 shows a foreign key defined on the DEPTNO column of the EMP table. It guarantees that every value in this column must match a value in the primary key of the DEPT table (also the DEPTNO column). Therefore, no erroneous department numbers can exist in the DEPTNO column of the EMP table.

Foreign keys can consist of multiple columns. However, a composite foreign key must reference a composite primary or unique key with the same number of columns and the same datatypes. Because composite primary and unique keys are limited to 32 columns, a composite foreign key is also limited to 32 columns.

Figure 24–6 Referential Integrity Constraints

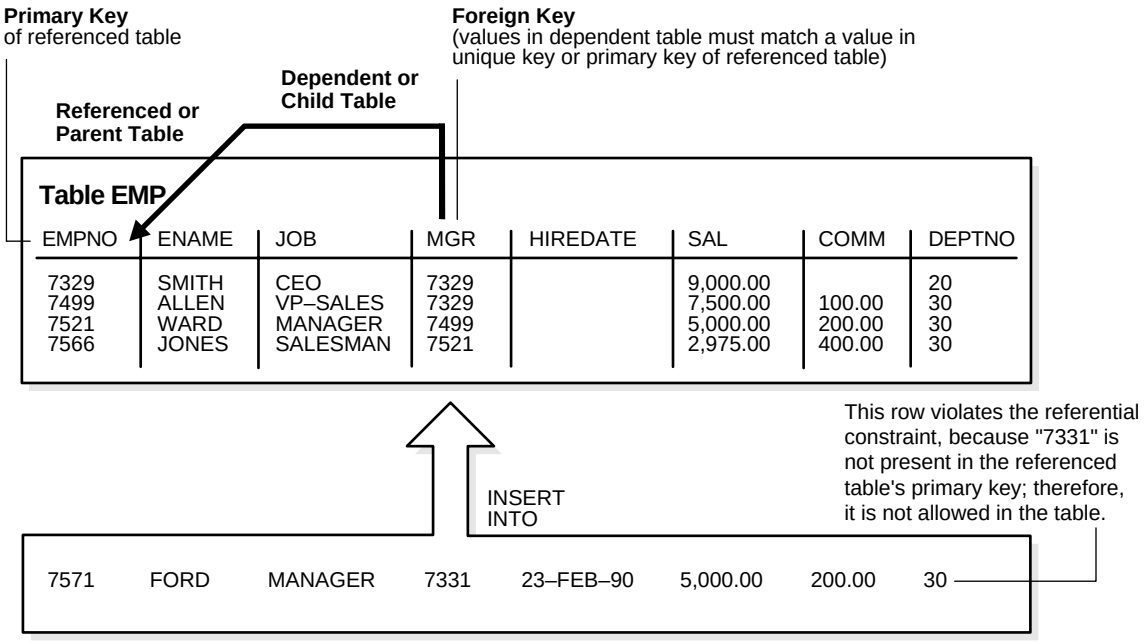


Self-Referential Integrity Constraints

Another type of referential integrity constraint, shown in Figure 24–7, is called a self-referential integrity constraint. This type of foreign key references a parent key in the same table.

In the example in Figure 24–7, the referential integrity constraint ensures that every value in the MGR column of the EMP table corresponds to a value that currently exists in the EMPNO column of the same table, but not necessarily in the same row (that is, every manager must also be an employee). This integrity constraint eliminates the possibility of erroneous employee numbers in the MGR column.

Figure 24–7 Single Table Referential Constraints



Nulls and Foreign Keys

The relational model permits the value of foreign keys either to match the referenced primary or unique key value, or be null. Several interpretations of this basic rule of the relational model are possible when composite (multicolumn) foreign keys are involved.

The ANSI/ISO SQL92 (entry-level) standard permits a composite foreign key to contain any value in its non-null columns if any other column is null, even if those non-null values are not found in the referenced key. By using other constraints (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 can specify particular actions to be performed on the dependent rows in a child table if a referenced parent key value is modified. The referential actions supported by the FOREIGN KEY integrity constraints of Oracle are UPDATE and DELETE No Action, and DELETE CASCADE.

Note: Other referential actions not supported by FOREIGN KEY integrity constraints of Oracle can be enforced using database triggers. See Chapter 18, “Database Triggers” for more information.

Update and Delete No Action The No Action (default) option specifies that referenced key values cannot be updated or deleted if the resulting data would violate a referential integrity constraint. For example, if a primary key value is referenced by a value in the foreign key, 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 — 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 24–1 outlines the DML statements allowed by the different referential actions on the primary / unique key values in the parent table, and the foreign key values in the child table.

Table 24–1 DML Statements Allowed by Update and Delete No Action

DML Statement	Issued Against Parent Table	Issued Against Child Table
INSERT	Always OK if the parent key value is unique.	OK only if the foreign key value exists in the parent key or is partially or all null.
UPDATE No Action	Allowed if the statement does not leave any rows in the child table without a referenced parent key value.	Allowed if the new foreign key value still references a referenced key value.
DELETE No Action	Allowed if no rows in the child table reference the parent key value.	Always OK.
DELETE Cascade	Always OK.	Always OK.

CHECK Integrity Constraints

A CHECK integrity constraint on a column or set of columns requires that a specified condition be true or unknown for every row of the table. If a DML statement results in the condition of the CHECK constraint evaluating to false, the statement is rolled back.

The Check Condition

CHECK constraints enable you to enforce very specific or sophisticated integrity rules by specifying a check condition. The condition of a CHECK constraint has some limitations:

- it must be a Boolean expression evaluated using the values in the row being inserted or updated, and
- it cannot contain subqueries, sequences, the SQL functions SYSDATE, UID, USER, or USERENV, or the pseudocolumns LEVEL or ROWNUM.

In evaluating CHECK constraints that contain string literals or SQL functions with NLS parameters as arguments (such as TO_CHAR, TO_DATE, and TO_NUMBER), Oracle uses the database's NLS settings by default. You can override the defaults by specifying NLS parameters explicitly in such functions within the CHECK constraint definition.

> **Additional Information:** See the *Oracle8 Reference* for more information on NLS features.

Multiple CHECK Constraints

A single column can have multiple CHECK constraints that reference the column in its definition. There is no limit to the number of CHECK constraints that you can define on a column.

The Mechanisms of Constraint Checking

To know what types of actions are permitted when constraints are present, it is useful to understand when Oracle actually performs the checking of constraints. To illustrate this, an example or two is helpful. Assume the following:

- The EMP table has been defined as in Figure 24–7 on on page 24-14.
- 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 addresses only the EMPNO and MGR columns of the EMP table.

Consider the insertion of the first row into the EMP table. No rows currently exist, so how can a row be entered if the value in the MGR column cannot reference any existing value in the EMPNO column?

Three possibilities for doing this are:

- A null can be entered for the MGR column of the first row, assuming that the MGR column does not have a NOT NULL constraint defined on it. Because nulls are allowed in foreign keys, this row is inserted successfully into the table.
- The same value can be entered in both the EMPNO and MGR columns. This case reveals that Oracle performs its constraint checking *after* the statement has been completely executed. To allow a row to be entered with the same values in the parent key and the foreign key, Oracle must first execute the statement (that is, insert the new row) and then check to see if any row in the table has an EMPNO that corresponds to the new row's MGR.
- A multiple row INSERT statement, such as an INSERT statement with nested SELECT statement, can insert rows that reference one another. For example, the

first row might have EMPNO as 200 and MGR as 300, while the second row might have EMPNO as 300 and MGR as 200.

This case also shows that constraint checking is deferred until the complete execution of the statement; all rows are inserted first, then all rows are checked for constraint violations. (You can also defer the checking of constraints until the end of the *transaction*; see "Deferred Constraint Checking" on page 24-19.)

Consider the same self-referential integrity constraint in the following scenario:

- The company has been sold. Because of this sale, all employee numbers must be updated to be the current value plus 5000 to coordinate with the new company's employee numbers. Because manager numbers are really employee numbers, these values must also increase by 5000.

The table currently exists as illustrated in Figure 24–8.

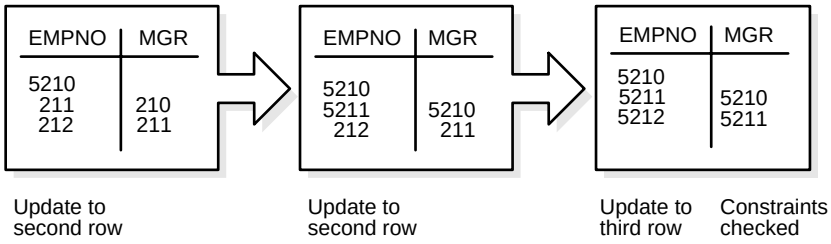
Figure 24–8 The EMP Table Before Updates

EMPNO	MGR
210	
211	210
212	211

```
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 24–9 shows that Oracle performs the actions of the entire SQL statement before any constraints are checked.

Figure 24–9 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. 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.

Deferred Constraint Checking

You can *defer* checking constraints for validity until the end of the transaction.

- A constraint is *deferred* if the system checks that it is satisfied only on commit. If a deferred constraint is violated, then commit causes the transaction to roll back.
- If a constraint is *immediate* (not deferred), then it is checked at the end of each statement. If it is violated, the statement is rolled back immediately.

If a constraint causes an *action* (for example, delete cascade), that action is always taken as part of the statement that caused it, whether the constraint is deferred or immediate.

Constraint Attributes

You can define constraints as either *deferrable* or *not deferrable*, and either *initially deferred* or *initially immediate*. These attributes can be different for each constraint. You specify them with keywords in the CONSTRAINT clause:

- DEFERRABLE or NOT DEFERRABLE
- INITIALLY DEFERRED or INITIALLY IMMEDIATE

> **Additional Information:** See *Oracle8 SQL Reference* for information about these constraint attributes and their default values.

Constraints can be added, dropped, enabled, disabled, or validated, but not altered. Specifically, you cannot alter a not-deferrable constraint to make it deferrable.

SET CONSTRAINTS Mode

The SET CONSTRAINTS statement makes constraints either DEFERRED or IMMEDIATE for a particular transaction (following the ANSI SQL92 standards in both syntax and semantics). You can use this statement to set the mode for a list of constraint names or for ALL constraints.

The SET CONSTRAINTS mode lasts for the duration of the transaction or until another SET CONSTRAINTS statement resets the mode.

SET CONSTRAINTS ... IMMEDIATE causes the specified constraints to be checked immediately on execution of each constrained statement. Oracle first checks any constraints that were deferred earlier in the transaction and then continues immediately checking constraints of any further statements in that transaction (as long as all the checked constraints are consistent and no other SET CONSTRAINTS statement is issued). If any constraint fails the check, an error is signalled; at that point, a COMMIT would cause the whole transaction to roll back.

The ALTER SESSION statement also has options to SET CONSTRAINTS IMMEDIATE or DEFERRED. These options imply setting ALL deferrable constraints (that is, you cannot specify a list of constraint names). They are equivalent to making a SET CONSTRAINTS statement at the start of each transaction in the current session.

Making constraints *immediate* at the end of a transaction is a way of checking whether COMMIT can succeed. You can avoid unexpected rollbacks by setting constraints to IMMEDIATE as the last statement in a transaction. If any constraint fails the check, you can then correct the error before committing the transaction.

The SET CONSTRAINTS statement is disallowed inside of triggers.

SET CONSTRAINTS can be a distributed statement. Existing database links that have transactions in process are told when a SET CONSTRAINTS ALL statement occurs, and new links learn that it occurred as soon as they start a transaction.

Unique Constraints and Indexes

A user will see inconsistent constraints, including duplicates in unique indexes, when that user's transaction produces these inconsistencies.

You can place deferred unique and foreign key constraints on snapshots, allowing fast and complete refresh to complete successfully.

Deferrable unique constraints always use nonunique indexes. When you remove a deferrable constraint, its index remains. (This is convenient because the storage information remains available after you disable a constraint.) Not-deferrable unique constraints and primary keys also use a nonunique index if the nonunique index is placed on the key columns before the constraint is enforced.

Enabled, Disabled, and Enable Novalidate Constraints

You can enable or disable integrity constraints at the table level using the CREATE TABLE or ALTER TABLE statement. The ENABLE NOVALIDATE option of the ALTER TABLE statement resumes constraint checking on disabled constraints without first validating all data in the table.

- ENABLE CONSTRAINT ensures that all rows in the table are valid, that is, all rows conform to the constraint.
- DISABLE CONSTRAINT allows the table to contain rows which violate the constraint.
- ENABLE NOVALIDATE CONSTRAINT allows existing rows to violate the constraint, but ensures that all new or modified rows are valid.

> **Additional Information:** See *Oracle8 Administrator's Guide* for more information about how to use the ENABLE, DISABLE, and ENABLE NOVALIDATE CONSTRAINT options.

25

Controlling 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*

This chapter explains how to control access to an Oracle database. It includes:

- Database Security
- Schemas, Database Users, and Security Domains
- User Authentication
- User Tablespace Settings and Quotas
- The User Group PUBLIC
- User Resource Limits and Profiles
- Licensing

> **Additional Information:** If you are using Trusted Oracle, see your *Trusted Oracle* documentation for information on database access in that environment.

Database Security

Database security entails allowing or disallowing user actions on the database and the objects within it. Oracle uses schemas and security domains to control access to data and to restrict the use of various database resources.

Oracle provides comprehensive discretionary access control. *Discretionary access control* regulates all user access to named objects through privileges. A privilege is permission to access a named object in a prescribed manner; for example, permission to query a table. Privileges are granted to users at the discretion of other users — hence the term "discretionary access control". For more information about privileges, see Chapter 26, "Privileges and Roles".

Schemas, Database Users, and Security Domains

A *user* (sometimes called a *username*) is a name defined in the database that can connect to and access objects. A *schema* is a named collection of objects, such as tables, views, clusters, procedures, and packages, associated with a particular user. Schemas and users help database administrators manage database security.

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 often used interchangeably.

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 on 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 first four security domain options listed above; privileges and roles are discussed in Chapter 26, “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.

Additional Information: See the *Oracle8 Administrator’s Guide* for more information about the special users SYS and SYSTEM.

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 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 network authentication.

Oracle requires special authentication procedures for database administrators, because they perform special database operations.

Authentication by the Operating System

Some operating systems permit Oracle to use information maintained by the operating system to authenticate users. The benefits of authentication by the operating system are:

- Users can connect to Oracle more conveniently (without specifying a username or password). For example, a user can invoke SQL*Plus and skip the username and password prompts by entering

  ```
  SQLPLUS /
  ```

- Control over user authorization is centralized in the operating system; Oracle need not store or manage user passwords. However, Oracle still maintains usernames in the database.
- Username entries in the database and operating system audit trails correspond.

If the operating system is used to authenticate database users, some special considerations arise with respect to distributed database environments and database links; see Chapter 30, "Distributed Databases", for information on this topic.

> **Additional Information:** See your Oracle operating system-specific documentation for more information about authenticating via your operating system.

Authentication by the Network

If network authentication services are available to you (such as DCE, Kerberos, or SESAME), Oracle can accept authentication from the network service. To use a network authentication service with Oracle, you must also have the Oracle Secure Network Services product.

> **Additional Information:** If you use a network authentication service, some special considerations arise for network roles and database links. See *Oracle8 Distributed Database Systems* for more information about network authentication.

Authentication by 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 in an encrypted format. A user can change his or her password at any time.

Password Encryption while Connecting

To protect password confidentiality, Oracle allows you to encrypt passwords during network (client / server and server / server) connections. If you enable this functionality on the client and server machines, Oracle encrypts passwords using a

modified DES (Data Encryption Standards) algorithm before sending them across the network.

> **Additional Information:** See *Oracle8 Distributed Database Systems* for more information about encrypting passwords in network systems.

Account Locking

Oracle can lock a user's account if the user fails to login to the system within a specified number of attempts. Depending on how the account is configured, it can be unlocked automatically after a specified time interval or it must be unlocked by the database administrator.

The CREATE PROFILE statement configures the number of failed logins a user can attempt and the amount of time the account remains locked before automatic unlock. See "Profiles" on page 25-13 for information about profiles.

The database administrator can also lock accounts manually. When this occurs, the account cannot be unlocked automatically but must be unlocked explicitly by the database administrator.

Password Lifetime and Expiration

Password lifetime and expiration options allow the database administrator to specify a lifetime for passwords, after which time they expire and must be changed before a login to the account can be completed. On first attempt to login to the database account after the password expires, the user's account enters the grace period, and a warning message is issued to the user every time the user tries to login until the grace period is over.

The user is expected to change the password within the grace period. If the password is not changed within the grace period, the account is locked and no further logins to that account are allowed without assistance by the database administrator.

The database administrator can also set the password state to expired. When this happens, the users account status is changed to expired, and when the user logs in, the account enters the grace period.

Password History

The password history option checks each newly specified password to ensure that a password is not reused for the specified amount of time or for the specified number of password changes. The database administrator can configure the rules for password reuse with CREATE PROFILE statements.

Password Complexity Verification

Complexity verification checks that each password is complex enough to provide reasonable protection against intruders who try to break into the system by guessing passwords.

The Oracle default password complexity verification routine requires that each password:

- be a minimum of four characters in length
- not equal the userid
- include at least one alphabet character, one numeric character, and one punctuation mark
- not match any word on an internal list of simple words like welcome, account, database, user, and so on.
- differ from the previous password by at least three characters.

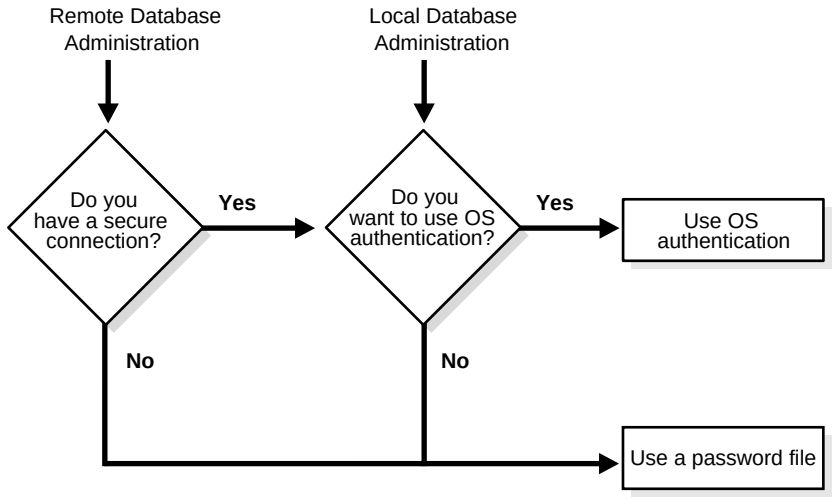
Database Administrator Authentication

Database administrators perform special operations (such as shutting down or starting up a database) that should not be performed by normal database users. Oracle provides a more secure authentication scheme for database administrator usernames.

You can choose between operating system authentication or password files to authenticate database administrators.

Figure 25–1 illustrates the choices you have for database administrator authentication schemes, depending on whether you administer your database locally (on the same machine on which the database resides) or if you administer many different database machines from a single remote client.

Figure 25–1 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.

> **Additional Information:** See your Oracle operating-system-specific documentation for information about OS authentication of database administrators.

The database uses password files to keep track of database usernames who have been granted the SYSDBA and SYSOPER privileges. These privileges allow database administrators to perform the following actions:

SYSOPER	Permits you to perform STARTUP, SHUTDOWN, ALTER DATABASE OPEN / MOUNT, ALTER DATABASE BACKUP, ARCHIVE LOG, and RECOVER, and includes the RESTRICTED SESSION privilege.
SYSDBA	Contains all system privileges with ADMIN OPTION, and the SYSOPER system privilege; permits CREATE DATABASE and time-based recovery.

> **Additional Information:** See the *Oracle8 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 without specifying a tablespace to contain the object, Oracle places the object in the user's default tablespace. You set a user's default tablespace when the user is created; you can change it after the user has been created.

Temporary Tablespace

When a user executes a SQL statement that requires the creation of a temporary segment, Oracle allocates that segment in the user's temporary tablespace.

Tablespace Access and Quotas

You can assign to each user a *tablespace quota* for any tablespace of the database. Doing so can accomplish two things:

- You allow the user to use the specified tablespace to create objects, provided that the user has the appropriate privileges.
- You can limit the amount of space allocated for storage of a the's objects in the specified tablespace.

By default, each user has no quota on any tablespace in the database. Therefore, if the user has the privilege to create some type of schema object, he or she must also have been either assigned a tablespace quota in which to create the object or been given the privilege to create that object in the schema of another user who was assigned a sufficient tablespace quota.

You can assign two types of tablespace quotas to a user: a quota for a specific amount of disk space in the tablespace (specified in bytes, kilobytes, or megabytes), or a quota for an unlimited amount of disk space in the tablespace. You should assign specific quotas to prevent a user's objects from consuming too much space in a tablespace.

Tablespace quotas and temporary segments have no effect on each other:

- Temporary segments do not consume any quota that a user might possess.
- Temporary segments can be created in a tablespace for which a user has no quota.

You can assign a tablespace quota to a user when you create that user, and you can change that quota or add a different quota later.

Revoke a user's tablespace access by altering the user's current quota to zero. With a quota of zero, the user's objects in the revoked tablespace remain, but the objects cannot be allocated any new space.

The User Group PUBLIC

Each database contains a user group called PUBLIC. The PUBLIC user group provides public access to specific schema objects (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 26, "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 some system and object privileges to and from PUBLIC can cause every view, procedure, function, package, and trigger in the database to be recompiled.

PUBLIC has the following restrictions:

- You cannot assign tablespace quotas to PUBLIC, although you can assign the UNLIMITED TABLESPACE system privilege to PUBLIC.
- You can create database links and synonyms as PUBLIC (using CREATE PUBLIC DATABASE LINK/SYNONYM), but no other schema object can be owned by PUBLIC. For example, the following statement is not legal:

  ```
  CREATE TABLE public.emp . . . ;
  ```

Note: Rollback segments can be created with the keyword PUBLIC, but these are not owned by the PUBLIC user group. All rollback segments are owned by SYS. See Chapter 2, "Data Blocks, Extents, and Segments", for information about rollback segments.

User Resource Limits and Profiles

You can set limits on the amount of various system resources available to each user as part of a user's security domain. By doing so, you can prevent the uncontrolled consumption of valuable system resources such as CPU time.

This resource limit feature is very useful in large, multiuser systems, where system resources are very expensive. Excessive consumption of these resources by one or more users can detrimentally affect the other users of the database. In single-user or small-scale multiuser database systems, the system resource feature is not as important, because users' consumption of system resources is less likely to have detrimental impact.

You manage a user's resource limits with his or her profile—a named set of resource limits that you can assign to that user. Each Oracle database can have an unlimited number of profiles. Oracle allows the security administrator to enable or disable the enforcement of profile resource limits universally.

If you set resource limits, a slight degradation in performance occurs when users create sessions. This is because Oracle loads all resource limit data for the user when a user connects to a database.

Types of System Resources and Limits

Oracle can limit the use of several types of system resources, including CPU time and logical reads. In general, you can control each of these resources at the session level, the call level, or both:

Session Level — Each time a user connects to a database, a session is created. Each session consumes CPU time and memory on the computer that executes Oracle. You can set several resource limits at the session level.

If a user exceeds a session-level resource limit, Oracle terminates (rolls back) the current statement and returns a message indicating the session limit has been reached. At this point, all previous statements in the current transaction are intact, and the only operations the user can perform are COMMIT, ROLLBACK, or disconnect (in this case, the current transaction is committed); all other operations produce an error. Even after the transaction is committed or rolled back, the user can accomplish no more work during the current session.

Call Level

Each time a SQL statement is executed, several steps are taken to process the statement. During this processing, several calls are made to the database as part of the different execution phases. To prevent any one call from using the system excessively, Oracle allows you to set several resource limits at the call level.

If a user exceeds a call-level resource limit, Oracle halts the processing of the statement, rolls back the statement, and returns an error. However, all previous statements of the current transaction remain intact, and the user's session remains connected.

CPU Time

When SQL statements and other types of calls are made to Oracle, an amount of CPU time is necessary to process the call. Average calls require a small amount of CPU time. However, a SQL statement involving a large amount of data or a runaway query can potentially consume a large amount of CPU time, reducing CPU time available for other processing.

To prevent uncontrolled use of CPU time, you can limit the CPU time per call and the total amount of CPU time used for Oracle calls during a session. The limits are set and measured in CPU one-hundredth seconds (0.01 seconds) used by a call or a session.

Logical Reads

Input/output (I/O) is one of the most expensive operations in a database system. SQL statements that are I/O intensive can monopolize memory and disk use and cause other database operations to compete for these resources.

To prevent single sources of excessive I/O, Oracle let you limit the logical data block reads per call and per session. Logical data block reads include data block reads from both memory and disk. The limits are set and measured in number of block reads performed by a call or during a session.

Other Resources

Oracle also provides for the limitation of several other resources at the session level:

- You can limit the number of *concurrent sessions per user*. Each user can create only up to a predefined number of concurrent sessions.
- You can limit the *idle time* for a session. If the time between Oracle calls for a session reaches the idle time limit, the current transaction is rolled back, the session is aborted, and the resources of the session are returned to the system. The next call receives an error that indicates the user is no longer connected to the instance. This limit is set as a number of elapsed minutes.

  Note: Shortly after a session is aborted because it has exceeded an idle time limit, the process monitor (PMON) background process cleans up after the aborted session. Until PMON completes this process, the aborted session is still counted in any session/user resource limit.

- You can limit the elapsed connect time per session. If a session's duration exceeds the elapsed time limit, the current transaction is rolled back, the session is dropped, and the resources of the session are returned to the system. This limit is set as a number of elapsed minutes.

  Note: Oracle does not constantly monitor the elapsed idle time or elapsed connection time. Doing so would reduce system performance. Instead, it checks every few minutes. Therefore, a session can exceed this limit slightly (for example, by five minutes) before Oracle enforces the limit and aborts the session.

- You can limit the amount of private SGA space (used for private SQL areas) for a session. This limit is only important in systems that use the multithreaded server configuration; otherwise, private SQL areas are located in the PGA. This limit is set as a number of bytes of memory in an instance's SGA. Use the characters "K" or "M" to specify kilobytes or megabytes.

Additional Information: Instructions on enabling and disabling resource limits are included in the *Oracle8 Administrator's Guide*.

Profiles

A profile is a named set of specified resource limits that can be assigned to valid username of an Oracle database. Profiles provide for easy management of resource limits.

When to Use Profiles

You need to create and manage user profiles only if resource limits are a requirement of your database security policy. To use profiles, first categorize the related types of users in a database. Just as roles are used to manage the privileges of related users, profiles are used to manage the resource limits of related users. Determine how many profiles are needed to encompass all types of users in a database and then determine appropriate resource limits for each profile.

Determining Values for Resource Limits of a Profile

Before creating profiles and setting the resource limits associated with them, you should determine appropriate values for each resource limit. You can base these values on the type of operations a typical user performs. For example, if one class of user does not normally perform a high number of logical data block reads, then the LOGICAL_READS_PER_SESSION and LOGICAL_READS_PER_CALL limits should be set conservatively.

Usually, the best way to determine the appropriate resource limit values for a given user profile is to gather historical information about each type of resource usage. For example, the database or security administrator can use the AUDIT SESSION option to gather information about the limits CONNECT_TIME, LOGICAL_READS_PER_SESSION, and LOGICAL_READS_PER_CALL. See Chapter 27, "Auditing", for more information.

You can gather statistics for other limits using the Monitor feature of Oracle Enterprise Manager (or Server Manager), specifically the Statistics monitor.

Licensing

Oracle is usually licensed for use by a maximum number of named users or by a maximum number of concurrently connected users. The database administrator (DBA) is responsible for ensuring that the site complies with its license agreement. Oracle's licensing facility helps the DBA monitor system use by tracking and limiting the number of sessions concurrently connected to an instance or the number of users created in a database.

If the DBA discovers that more than the licensed number of sessions need to connect, or more than the licensed number of users need to be created, he or she can upgrade the Oracle license to raise the appropriate limit. (To upgrade an Oracle license, you must contact your Oracle representative.)

Note: When Oracle is embedded in an Oracle application (such as Oracle Office), run on some older operating systems, or purchased for use in some countries, it is not licensed for either a set number of sessions or a set group of users. In such cases only, the Oracle licensing mechanisms do not apply and should remain disabled.

The following sections explain the two major types of licensing available for Oracle.

Additional Information: See the *Oracle8 Administrator's Guide* for more information about licensing.

Concurrent Usage Licensing

In *concurrent usage licensing*, the license specifies a number of *concurrent users*, which are sessions that can be connected concurrently to the database on the specified computer at any time. This number includes all batch processes and online users. If a single user has multiple concurrent sessions, each session counts separately in the total number of sessions. If multiplexing software (such as a TP monitor) is used to reduce the number of sessions directly connected to the database, the number of concurrent users is the number of distinct inputs to the multiplexing front end.

The concurrent usage licensing mechanism allows a DBA to:

- Set a limit on the number of concurrent sessions that can connect to an instance by setting the LICENSE_MAX_SESSIONS parameter. Once this limit is reached, only users who have the RESTRICTED SESSION system privilege can

connect to the instance; this allows DBA to kill unneeded sessions, allowing other sessions to connect.

- Set a warning limit on the number of concurrent sessions that can connect to an instance by setting the LICENSE_SESSIONS_WARNING parameter. Once the warning limit is reached, Oracle allows additional sessions to connect (up to the maximum limit described above), but sends a warning message to any user who connects with RESTRICTED SESSION privilege and records a warning message in the database's ALERT file.

The DBA can set these limits in the database's parameter file so that they take effect when the instance starts and can change them while the instance is running (using the ALTER SYSTEM command). The latter is useful for databases that cannot be taken offline.

The session licensing mechanism allows a DBA to check the current number of connected sessions and the maximum number of concurrent sessions since the instance started. The V$LICENSE view shows the current settings for the license limits, the current number of sessions, and the highest number of concurrent sessions since the instance started (the session "high water mark"). The DBA can use this information to evaluate the system's licensing needs and plan for system upgrades.

For instances running with the Oracle Parallel Server, each instance can have its own concurrent usage limit and warning limit. The sum of the instances' limits must not exceed the site's concurrent usage license.

The concurrent usage limits apply to all user sessions, including sessions created for incoming database links. They do not apply to sessions created by Oracle or to recursive sessions. Sessions that connect through external multiplexing software are not counted separately by the Oracle licensing mechanism, although each contributes individually to the Oracle license total. The DBA is responsible for taking these sessions into account.

Named User Licensing

In *named user licensing*, the license specifies a number of named users, where a *named user* is an individual who is authorized to use Oracle on the specified computer. No limit is set on the number of sessions each user can have concurrently, or on the number of concurrent sessions for the database.

Named user licensing allows a DBA to set a limit on the number of users that are defined in a database, including users connected via database links. Once this limit is reached, no one can create a new user. This mechanism assumes that each person

accessing the database has a unique user name in the database and that no two (or more) people share a user name.

The DBA can set this limit in the database's parameter file so that it takes effect when the instance starts and can change it while the instance is running (using the ALTER SYSTEM command). The latter is useful for databases that cannot be taken offline.

If multiple instances connect to the same database in an Oracle Parallel Server, all instances connected to the same database should have the same named user limit.

> **Additional Information:** See *Oracle8 Parallel Server Concepts and Administration* for more information on the Oracle Parallel Server.

26

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 you can control users' ability to execute system operations and to access schema objects by using privileges and roles. The chapter includes:

- Privileges
  - System Privileges
  - Schema Object Privileges
- Roles

> **Additional Information:** If you are using Trusted Oracle, see your *Trusted Oracle* documentation for information about roles and privileges in that environment.

Privileges

A *privilege* is a right to execute a particular type of SQL statement or to access another user's object. Some examples of privileges include the right to

- connect to the database (create a session)
- create a table
- select rows from another user's table
- execute another user's stored procedure

You grant privileges to users so these users can accomplish tasks required for their job. You should grant a privilege only to a user who absolutely requires the privilege to accomplish necessary work. Excessive granting of unnecessary privileges can compromise security. A user can receive a privilege in two different ways:

- You can grant privileges to users explicitly. For example, you can explicitly grant the privilege to insert records into the EMP table to the user SCOTT.
- You can also grant privileges to a role (a named group of privileges), and then grant the role to one or more users. For example, you can grant the privileges to select, insert, update, and delete records from the EMP table to the role named CLERK, which in turn you can grant to the users SCOTT and BRIAN.

Because roles allow for easier and better management of privileges, you should normally grant privileges to roles and not to specific users.

There are two distinct categories of privileges:

- system privileges
- schema object privileges

> **Additional Information:** Complete listings of all system and schema object privileges, as well as instructions for privilege management, appear in the *Oracle8 Administrator's Guide*.

System Privileges

A system privilege is the right to perform a particular action, or to perform an action on any schema objects of a particular type. For example, the privileges to create tablespaces and to delete the rows of any table in a database are system privileges. There are over 60 distinct system privileges.

Granting and Revoking System Privileges

You can grant or revoke system privileges to users and roles. If you grant system privileges to roles, you can use the roles to manage system privileges (for example, roles permit privileges to be made selectively available).

> **Note:** Usually, you should grant system privileges only to administrative personnel and application developers, because end users normally do not require the associated capabilities.

System privileges are granted to or revoked from users and roles using either of the following:

- the Grant System Privileges / Roles dialog box and Revoke System Privileges / Roles dialog box of Oracle Enterprise Manager
- the SQL commands GRANT and REVOKE

Who Can Grant or Revoke System Privileges?

Only users who have been granted a specific system privilege with the ADMIN OPTION or users with the GRANT ANY PRIVILEGE system privilege (typically database or security administrators) can grant or revoke system privileges to other users.

Schema Object Privileges

A schema object privilege ("*object privilege*") is a privilege or right to perform a particular action on a *specific* table, view, sequence, procedure, function, or package. Different object privileges are available for different types of schema objects. For example, the privilege to delete rows from the table DEPT is an object privilege.

Some schema objects (such as clusters, indexes, triggers, and database links) do not have associated object privileges; their use is controlled with system privileges. For example, to alter a cluster, a user must own the cluster or have the ALTER ANY CLUSTER system privilege.

A schema object and its synonym are equivalent with respect to privileges; that is, the object privileges granted for a table, view, sequence, procedure, function, or package apply whether referencing the base object by name or using a synonym.

For example, assume there is a table JWARD.EMP with a synonym named JWARD.EMPLOYEE and the user JWARD issues the following statement:

```
GRANT SELECT ON emp TO swilliams;
```

The user SWILLIAMS can query JWARD.EMP by referencing the table by name or using the synonym JWARD.EMPLOYEE:

```
SELECT * FROM jward.emp;
SELECT * FROM jward.employee;
```

If you grant object privileges on a table, view, sequence, procedure, function, or package to a *synonym* for the object, the effect is the same as if no synonym were used. For example, if JWARD wanted to grant the SELECT privilege for the EMP table to SWILLIAMS, JWARD could issue either of the following statements:

```
GRANT SELECT ON emp TO swilliams;
GRANT SELECT ON employee TO swilliams;
```

If a synonym is dropped, all grants for the underlying schema object remain in effect, even if the privileges were granted by specifying the dropped synonym.

Granting and Revoking Schema Object Privileges

Schema object privileges can be granted to and revoked from users and roles. If you grant object privileges to roles, you can make the privileges selectively available. Object privileges for users and roles can be granted or revoked using the SQL commands GRANT and REVOKE, respectively, or the Add Privilege to Role / User dialog box and Revoke Privilege from Role / User dialog box of Oracle Enterprise Manger.

Who Can Grant Schema Object Privileges?

A user automatically has all object privileges for schema objects contained in his or her schema. A user can grant any object privilege on any schema object he or she owns to any other user or role. If the grant includes the GRANT OPTION (of the GRANT command), the grantee can further grant the object privilege to other users; otherwise, the grantee can use the privilege but cannot grant it to other users.

Table Security Topics

Schema object privileges for tables allow table security at the level of DML and DDL operations.

Data Manipulation Language (DML) Operations The DELETE, INSERT, SELECT, and UPDATE privileges allow the DELETE, INSERT, SELECT, and UPDATE DML operations, respectively, on a table or view. You should grant these privileges only to users and roles that need to query or manipulate a table's data.

> **Additional Information:** See the *Oracle8 SQL Reference* for more information on these DML operations.

You can restrict INSERT and UPDATE privileges for a table to specific columns of the table. With selective INSERT, a privileged user can insert a row with values for the selected columns; all other columns receive NULL or the column's default value. With selective UPDATE, a user can update only specific column values of a row. Selective INSERT and UPDATE privileges are used to restrict a user's access to sensitive data.

For example, if you do not want data entry users to alter the SAL column of the employee table, selective INSERT and/or UPDATE privileges can be granted that exclude the SAL column. (Alternatively, a view that excludes the SAL column could satisfy this need for additional security.)

Data Definition Language (DDL) Operations The ALTER, INDEX, and REFERENCES privileges allow DDL operations to be performed on a table. Because these privileges allow other users to alter or create dependencies on a table, you should grant privileges conservatively. A user attempting to perform a DDL operation on a table may need additional system or object privileges (for example, to create a trigger on a table, the user requires both the ALTER TABLE object privilege for the table and the CREATE TRIGGER system privilege).

As with the INSERT and UPDATE privileges, the REFERENCES privilege can be granted on specific columns of a table. The REFERENCES privilege enables the grantee to use the table on which the grant is made as a parent key to any foreign keys that the grantee wishes to create in his or her own tables. This action is controlled with a special privilege because the presence of foreign keys restricts the data manipulation and table alterations that can be done to the parent key. A column-specific REFERENCES privilege restricts the grantee to using the named columns (which, of course, must include at least one primary or unique key of the parent table). See Chapter 24, "Data Integrity" for more information about primary keys, unique keys, and integrity constraints.

View Security Topics

Schema object privileges for views allow various DML operations, which actually affect the base tables from which the view is derived. DML object privileges for tables can be applied similarly to views.

Privileges Required to Create Views To create a view, you must meet the following requirements:

- You must have been granted 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 through a role.
- You must have been explicitly granted the SELECT, INSERT, UPDATE, or DELETE object privileges on all base objects underlying the view or the SELECT ANY TABLE, INSERT ANY TABLE, UPDATE ANY TABLE, or DELETE ANY TABLE system privileges. You may *not* have obtained these privileges through roles.
- Additionally, in order to grant other users access to your view, you must have received object privilege(s) to the base objects with the GRANT OPTION option or appropriate system privileges with the ADMIN OPTION option. If you have not, grantees cannot access your view.

Increasing Table Security Using Views To use a view, you require appropriate privileges only for the view itself. You do not require privileges on base object(s) underlying the view.

Views add two more levels of security for tables, column-level security and value-based security:

- A view can provide access to selected columns of base table(s). For example, you can define a view on the EMP table to show only the EMPNO, ENAME, and MGR columns:

```
CREATE VIEW emp_mgr AS
    SELECT ename, empno, mgr FROM emp;
```

- A view can provide value-based security for the information in a table. A WHERE clause in the definition of a view displays only selected rows of base tables. Consider the following two examples:

```
CREATE VIEW lowsal AS
    SELECT * FROM emp
    WHERE sal < 10000;
```

The LOWSAL view allows access to all rows of the EMP table that have a salary value less than 10000. Notice that all columns of the EMP table are accessible in the LOWSAL view.

```
CREATE VIEW own_salary AS
    SELECT ename, sal
    FROM emp
    WHERE ename = USER;
```

In the OWN_SALARY view, only the rows with an ENAME that matches the current user of the view are accessible. The OWN_SALARY view uses the USER pseudocolumn, whose values always refer to the current user. This view combines both column-level security and value-based security.

Procedure Security Topics

The one *schema 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 accesses. By writing a procedure and granting only EXECUTE privilege to a user, 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).

Procedure Execution and Security Domains A user with the EXECUTE object 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 privileges through roles to execute procedures.

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 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, resulting in tighter control of database access.

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 that user is privileged to execute. As a result of the SQL statement, a trigger is fired. The statements within the triggered action temporarily execute under the security domain of the user that owns the trigger.

System Privileges Needed to Create or Alter a Procedure To create a procedure, a user must have the CREATE PROCEDURE or CREATE ANY PROCEDURE *system privilege*. To alter a procedure, that is, to manually recompile a procedure, a user must own the procedure or have the ALTER ANY PROCEDURE system privilege.

The user who owns the procedure also must hav privileges for schema objects referenced in the procedure body. To create a procedure, you must have been explicitly granted the necessary privileges (system or object) on all objects referenced by the procedure; you cannot have obtained the required privileges through roles. This includes the EXECUTE privilege for any procedures that are called inside the procedure being created.

Triggers also require that privileges to referenced objects be granted explicitly to the trigger owner. Anonymous PL/SQL blocks can use any privilege, whether the privilege is granted explicitly or via a role.

Packages and Package Objects A user with the EXECUTE object privilege for a package can execute any (public) procedure or function in the package and access or modify the value of any (public) package variable. Specific EXECUTE privileges cannot be granted for a package's constructs. Therefore, you may find it useful to consider two alternatives for establishing security when developing procedures, functions, and packages for a database application. These alternatives are described in the following examples.

Example 1: This example shows four procedures created in the bodies of two packages.

```
CREATE PACKAGE BODY hire_fire AS
  PROCEDURE hire(...) IS
    BEGIN
      INSERT INTO emp . . .
    END hire;
  PROCEDURE fire(...) IS
    BEGIN
      DELETE FROM emp . . .
```

```
    END fire;
END hire_fire;

CREATE PACKAGE BODY raise_bonus AS
  PROCEDURE give_raise(...) IS
    BEGIN
      UPDATE EMP SET sal = . . .
    END give_raise;
  PROCEDURE give_bonus(...) IS
    BEGIN
      UPDATE EMP SET bonus = . . .
    END give_bonus;
END raise_bonus;
```

Access to execute the procedures is given by granting the EXECUTE privilege for the package, using the following statements:

```
GRANT EXECUTE ON hire_fire TO big_bosses;
GRANT EXECUTE ON raise_bonus TO little_bosses;
```

Granting EXECUTE privilege granted for a package provides uniform access to all package objects.

Example 2: This example shows four procedure definitions within the body of a single package. Two additional standalone procedures and a package are created specifically to provide access to the procedures defined in the main package.

```
CREATE PACKAGE BODY employee_changes AS
  PROCEDURE change_salary(...) IS BEGIN ... END;
  PROCEDURE change_bonus(...) IS BEGIN ... END;
  PROCEDURE insert_employee(...) IS BEGIN ... END;
  PROCEDURE delete_employee(...) IS BEGIN ... END;
END employee_changes;

CREATE PROCEDURE hire
  BEGIN
    employee_changes.insert_employee(...)
  END hire;

CREATE PROCEDURE fire
  BEGIN
    employee_changes.delete_employee(...)
  END fire;
```

```
PACKAGE raise_bonus IS
  PROCEDURE give_raise(...) AS
    BEGIN
      employee_changes.change_salary(...)
    END give_raise;

  PROCEDURE give_bonus(...)
    BEGIN
      employee_changes.change_bonus(...)
    END give_bonus;
```

Using this method, the procedures that actually do the work (the procedures in the EMPLOYEE_CHANGES package) are defined in a single package and can share declared global variables, cursors, on so on. By declaring top-level procedures HIRE and FIRE, and an additional package RAISE_BONUS, you can grant selective EXECUTE privileges on procedures in the main package:

```
GRANT EXECUTE ON hire, fire TO big_bosses;
GRANT EXECUTE ON raise_bonus TO little_bosses;
```

Roles

Oracle provides for easy and controlled privilege management through roles. *Roles* are named groups of related privileges that you grant to users or other roles. Roles are designed to ease the administration of end-user system and schema object privileges. However, roles are not meant to be used for application developers, because the privileges to access schema objects within stored programmatic constructs need to be granted directly. See “Data Definition Language Statements and Roles” on page 26-14 for more information about restrictions for procedures.

These properties of roles allow for easier privilege management within a database:

reduced privilege administration	Rather than granting the same set of privileges explicitly to several users, you can grant the privileges for a group of related users to a role, and then only the role needs to be granted to each member of the group.
dynamic privilege management	If the privileges of a group must change, only the privileges of the role need to be modified. The security domains of all users granted the group’s role automatically reflect the changes made to the role.

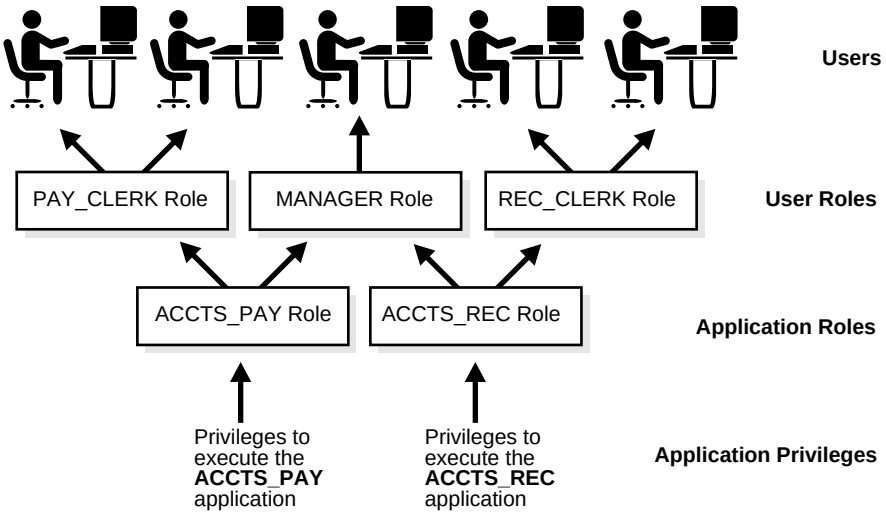
selective availability of privileges	You can selectively enable or disable the roles granted to a user. This allows specific control of a user's privileges in any given situation.
application awareness	The data dictionary records which roles exist, so you can design applications to query the dictionary and automatically enable (or disable) selective roles when a user attempts to execute the application by way of a given username.
application-specific security	You can protect role use with a password. Applications can be created specifically to enable a role when supplied the correct password. Users cannot enable the role if they do not know the password.

Additional Information: Instructions for enabling roles from an application are included in the *Oracle8 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 26–1 and the sections that follow describe the two uses of roles.

Figure 26–1 Common Uses for Roles



Application Roles

You grant an application role all privileges necessary to run a given database application. Then, you grant the application role to other roles or to specific users. An application can have several different roles, with each role assigned a different set of privileges that allow for more or less data access while using the application.

User Roles

You create a user role for a group of database users with common privilege requirements. You manage user privileges by granting application roles and privileges to the user role and then granting the user role to appropriate users.

The Mechanisms of Roles

Database roles have the following functionality:

- A role can be granted system or schema object privileges.
- A role can be granted to other roles. However, a role cannot be granted to itself and cannot be granted circularly (for example, role A cannot be granted to role B if role B has previously been granted to role A).
- Any role can be granted to any database user.
- Each role granted to a user is, at a given time, either enabled or disabled. A user's security domain includes the privileges of all roles currently enabled for the user and excludes the privileges of any roles currently disabled for the user. Oracle allows database applications and users to enable and disable roles to provide selective availability of privileges.
- An indirectly granted role (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 Grant System Privileges / Roles dialog box and Revoke System Privileges / Roles dialog box of Oracle Enterprise 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, or through network services.

> **Additional Information:** Detailed instructions on role management are included in the *Oracle8 Administrator's Guide*.

Who Can Grant or Revoke Roles?

Any user with the GRANT ANY ROLE system privilege can grant or revoke *any* role (except a global role) to or from other users or roles of the database. You should grant this system privilege conservatively because it is very powerful.

> **Additional Information:** See *Oracle8 Distributed Database Systems* for information about global roles.

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 Roles and Users

Each role and user has its own unique security domain. A role's security domain includes the privileges granted to the role plus those privileges granted to any roles that are granted to the role.

A user's security domain includes privileges on all schema objects in the corresponding schema, the privileges granted to the user, and the privileges of roles granted to the user that are *currently enabled*. (A role can be simultaneously enabled for one user and disabled for another.) A user's security domain also includes the privileges and roles granted to the user group PUBLIC.

Named PL/SQL Blocks and Roles

All roles are disabled in any named PL/SQL block (stored procedure, function, or trigger) that

- is created in a user schema that does not own the object being referenced in the PL/SQL block
- can be executed as a user other than the owner of the PL/SQL block

Anonymous PL/SQL blocks, however, are executed based on privileges granted through enabled roles.

The SESSION_ROLES view shows all roles that are currently enabled. If a named PL/SQL block queries SESSION_ROLES, the query does not return any rows.

Data Definition Language Statements and Roles

A user requires one or more privileges to successfully execute a data definition language (DDL) statement, depending on the statement. For example, to create a table, the user must have the CREATE TABLE or CREATE ANY TABLE system privilege. To create a view of another user's table, the creator requires the CREATE

VIEW or CREATE ANY VIEW system privilege and either the SELECT object privilege for the table or the SELECT ANY TABLE system privilege.

Oracle avoids the dependencies on privileges received by way of roles by restricting the use of specific privileges in certain DDL statements. The following rules outline these privilege restrictions concerning DDL statements:

- All system privileges and schema object privileges that permit a user to perform a DDL operation are usable when received through a role.

  Examples:

  – System Privileges: the CREATE TABLE, CREATE VIEW and CREATE PROCEDURE privileges.

  – Schema Object Privileges: the ALTER and INDEX privileges for a table.

  Exception: The REFERENCES object privilege for a table cannot be used to define a table's foreign key if the privilege is received through a role.

- All system privileges and object privileges that allow a user to perform a DML operation that is required to issue a DDL statement are *not* usable when received through a role.

  Example: If a user receives the SELECT ANY TABLE system privilege or the SELECT object privilege for a table through a role, he or 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 through 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, but the user is indirectly granted the SELECT object 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 both the EMP and DEPT tables.
- Although the user has both the CREATE VIEW and SELECT privilege for the EMP table (both through a role), the user cannot create a usable view on the EMP table, because the SELECT object privilege for the EMP table was granted through a role. Any views created will produce errors when accessed.

- The user can create a view on the DEPT table, because the user has the CREATE VIEW privilege (through a role) and the SELECT privilege for the DEPT table (directly).

Predefined Roles

The 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 granting (and revoking) of database roles and to manage their password authentication.

This capability is not available on all operating systems.

> **Additional Information:** 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 ensure that all needed roles are set as the default roles for a distributed (remote) session. You cannot enable roles when connecting to a remote database from within a local database session. For example, you cannot execute a remote procedure that attempts to enable a role at the remote site.

> **Additional Information:** For more information about distributed database environments, see *Oracle8 Distributed Database Systems.*

27

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
- Schema Object Auditing
- Focusing Statement, Privilege, and Schema Object Auditing

> **Additional Information:** If you are using Trusted Oracle, see your *Trusted Oracle* documentation for information about auditing in that environment.

Introduction to Auditing

Auditing is the monitoring and recording of selected user database actions. Auditing is normally used to

- investigate suspicious activity. For example, if an unauthorized user is deleting data from tables, the security administrator might decide to audit all connections to the database and all successful and unsuccessful deletions of rows from all tables in the database.
- monitor and gather data about specific database activities. For example, the database administrator can gather statistics about which tables are being updated, how many logical I/Os are performed, or how many concurrent users connect at peak times.

Auditing Features

This section outlines the features of the Oracle auditing mechanism.

Types of Auditing

Oracle supports three general types of auditing:

statement auditing	The selective auditing of SQL statements with respect to only the type of statement, not the specific schema objects on which it operates. Statement auditing options are typically broad, auditing the use of several types of related actions per option. For example, AUDIT TABLE tracks several DDL statements regardless of the table on which they are issued. You can set statement auditing to audit selected users or every user in the database.
privilege auditing	The selective auditing of the use of powerful system privileges to perform corresponding actions, such as AUDIT CREATE TABLE. Privilege auditing is more focused than statement auditing because it audits only the use of the target privilege. You can set privilege auditing to audit a selected user or every user in the database.
schema object auditing	The selective auditing of specific statements on a particular schema object, such as AUDIT SELECT ON EMP. Schema object auditing is very focused, auditing only a specific statement on a specific schema object. Schema object auditing always applies to all users of the database.

Focus of Auditing

Oracle allows audit options to be focused or broad. You can

- audit successful statement executions, unsuccessful statement executions, or both
- 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 information such as the operation that was audited, the user performing the operation, and the date and time of the operation. Audit records can be stored in either a data dictionary table, called the database audit trail, or an operating system audit trail.

The database audit trail is a single table named AUD$ in the SYS schema of each Oracle database's data dictionary. Several predefined views are provided to help you use the information in this table.

> **Additional Information:** Instructions for creating and using these views are found in the *Oracle8 Administrator's Guide*.

The audit trail records can contain different types of information, depending on the events audited and the auditing options set. The following information is always included in each audit trail record, provided that the information is meaningful to the particular audit action:

- the user name
- the session identifier
- the terminal identifier
- the name of the schema object accessed
- the operation performed or attempted
- the completion code of the operation
- the date and time stamp
- the system privileges used (including MAC privileges for Trusted Oracle)
- the label of the user session (for Trusted Oracle only)
- the label of the schema object accessed (for Trusted Oracle only)

The operating system audit trail is encoded and not readable, but it is decoded in data dictionary files and error messages as follows:

Action Code	This describes the operation performed or attempted. The AUDIT_ACTIONS data dictionary table contains a list of these codes and their descriptions.
Privileges Used	This describes any system privileges used to perform the operation. The SYSTEM_PRIVILEGE_MAP table lists all of these codes and their descriptions.
Completion Code	This describes the result of the attempted operation. Successful operations return a value of zero; unsuccessful operations return the Oracle error code describing why the operation was unsuccessful. These codes are listed in *Oracle8 Error Messages*.

Auditing Mechanisms

This section explains the mechanisms used by the Oracle auditing features.

When Are Audit Records Generated?

The recording of audit information can be enabled or disabled. This functionality allows any authorized database user to set audit options at any time but reserves control of recording audit information for the security administrator.

> **Additional Information:** Instructions on enabling and disabling auditing are found in the *Oracle8 Administrator's Guide*.

When 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 14, "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 with administrator privileges. Connections by these users bypass certain internal features of Oracle to allow specific administrative operations to occur (for example, database startup, shutdown, recovery, and so on).

Events Always Audited to the Operating System Audit Trail

Regardless of whether database auditing is enabled, Oracle always records some database-related actions into the operating system audit trail:

Instance startup	An audit record is generated that details the OS user starting the instance, the user's terminal identifier, the date and time stamp, and whether database auditing was enabled or disabled. This information is recorded into the 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, the user's terminal identifier, the date and time stamp.
Connections to the database with administrator privileges	An audit record is generated that details the OS user connecting to Oracle with administrator privileges. This provides accountability of users connected with administrator privileges.

On operating systems that do not make an audit trail accessible to Oracle, these audit trail records are placed in an Oracle audit trail file in the same directory as background process trace files.

> **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 or privilege audit options being set or changed. The modified statement or privilege audit options take effect only when the current session is ended and a new session is created. In contrast, changes to schema object audit options become effective for current sessions immediately.

Auditing in a Distributed Database

Auditing is site autonomous; an instance audits only the statements issued by directly connected users. A local Oracle node cannot audit actions that take place in a remote database. Because remote connections are established through the user account of a database link, the remote Oracle node audits the statements issued through the database link's connection. See Chapter 30, "Distributed Databases", for more information about distributed databases and database links.

Auditing to the OS Audit Trail

Both Oracle and Trusted Oracle allow audit trail records to be directed to an operating system audit trail if the operating system makes such an audit trail available to Oracle. On some other operating systems, these audit records are written to a file outside the database, with a format similar to other Oracle trace files.

> **Additional Information:** See your platform-specific Oracle documentation to see if this feature has been implemented on your operating system.

Both Oracle and Trusted 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 usual cause of this is that the operating system audit trail or the file system is full and unable to accept new records.

System administrators configuring OS auditing should ensure that the audit trail or the file system does not fill completely. Most operating systems provide administrators with sufficient information and warning to ensure this does not occur. Note, however, that configuring auditing to use the database audit trail removes this vulnerability, because the Oracle server prevents audited events from occurring if the audit trail is unable to accept the database audit record for the statement.

Statement Auditing

Statement auditing is the selective auditing of related groups of statements that fall into two categories:

- DDL statements, regarding a particular *type* of database structure or schema object, but not a specifically named structure or schema object (for example, AUDIT TABLE audits all CREATE and DROP TABLE statements)
- DML statements, regarding a particular *type* of database structure or schema object, but not a specifically named structure or schema object (for example, AUDIT SELECT TABLE audits all SELECT . . . FROM TABLE / VIEW / SNAPSHOT statements, regardless of the table, view, or snapshot)

Statement auditing can be broad or focused, auditing the activities of all database users or the activities of only a select list of database users.

Privilege Auditing

Privilege auditing is the selective auditing of the statements allowed using a system privilege. For example, auditing of the SELECT ANY TABLE system privilege audits users' statements that are executed using the SELECT ANY TABLE system privilege. You can audit the use of any system privilege.

In all cases of privilege auditing, owner privileges and schema object privileges are checked before system privileges. If the owner and schema object privileges suffice to permit the action, the action is not audited.

If similar statement and privilege audit options are both set, only a single audit record is generated. For example, if the statement 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.

Like statement auditing, privilege auditing can audit the activities of all database users or the activities of only a select list of database users.

Schema Object Auditing

Schema bject auditing is the selective auditing of specific DML statements (including queries) and GRANT and REVOKE statements for specific schema objects. Schema object auditing audits the operations permitted by schema object privileges, such as SELECT or DELETE statements on a given table, as well as the GRANT and REVOKE statements that control those privileges.

You can audit statements that reference tables, views, sequences, *standalone* stored procedures and functions, and packages (procedures in packages cannot be audited individually).

Statements that reference clusters, database links, indexes, or synonyms are not audited directly. However, you can audit access to these schema objects indirectly by auditing the operations that affect the base table.

Schema object audit options are always set for all users of the database; these options cannot be set for a specific list of users. You can set default schema object audit options for all auditable schema objects.

> **Additional Information:** See "AUDIT (Schema Objects)" in *Oracle8 SQL Reference*.

Schema Object Audit Options for Views and Procedures

Views and procedures (including stored functions, packages, and triggers) reference underlying schema objects in their definition. Therefore, auditing with respect to views and procedures has several unique characteristics. Multiple audit records can be generated as the result of using a view or a procedure: The use of the view or procedure is subject to enabled audit options; and the SQL statements issued as a result of using the view or procedure are subject to the enabled audit options of the base schema objects (including default audit options).

Consider the following series of SQL statements:

```
AUDIT SELECT ON emp;

CREATE VIEW emp_dept AS
  SELECT empno, ename, dname
    FROM emp, dept
    WHERE emp.deptno = dept.deptno;

AUDIT SELECT ON emp_dept;

SELECT * FROM emp_dept;
```

As a result of the query on EMP_DEPT, two audit records are generated: one for the query on the EMP_DEPT view and one for the query on the base table EMP (indirectly through the EMP_DEPT view). The query on the base table DEPT does not generate an audit record because the SELECT audit option for this table is not enabled. All audit records pertain to the user that queried the EMP_DEPT view.

The audit options for a view or procedure are determined when the view or procedure is first used and placed in the shared pool. These audit options remain set until the view or procedure is flushed from, and subsequently replaced in, the shared pool. Auditing a schema object invalidates that schema object in the cache and causes it to be reloaded. Any changes to the audit options of base schema objects are not observed by views and procedures in the shared pool.

Continuing with the above example, if auditing of SELECT statements is turned off for the EMP table, use of the EMP_DEPT view would no longer generate an audit record for the EMP table.

Focusing Statement, Privilege, and Schema Object Auditing

Oracle allows you to focus statement, privilege, and schema object auditing in three areas:

- successful and unsuccessful executions of the audited SQL statement
- BY SESSION and BY ACCESS auditing
- for specific users or for all users in the database (statement and privilege auditing only)

Auditing Successful and Unsuccessful Statement Executions

For statement, privilege, and schema object auditing, Oracle allows the selective auditing of successful executions of statements, unsuccessful attempts to execute statements, or both. Therefore, you can monitor actions even if the audited statements do not complete successfully.

You can audit an unsuccessful statement execution only if a valid SQL statement is issued but fails because of lack of proper authorization or because it references a nonexistent schema object. Statements that failed to execute because they simply were not valid cannot be audited. For example, an enabled privilege auditing option set to audit unsuccessful statement executions audits statements that use the target system privilege but have failed for other reasons (such as when CREATE TABLE is set but a CREATE TABLE statement fails due to lack of quota for the specified tablespace).

Using either form of the AUDIT 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. This section describes the distinction between the BY SESSION and BY ACCESS options of the AUDIT command.

BY SESSION

For any type of audit (schema object, statement, or privilege), BY SESSION inserts only one audit record in the audit trail, per user and schema object, during the session that includes an audited action.

A *session* is the time between when a user connects to and disconnects from an Oracle database.

Example 1 Assume the following:

- The SELECT TABLE statement auditing option is set BY SESSION.
- JWARD connects to the database and issues five SELECT statements against the table named DEPT and then disconnects from the database.
- SWILLIAMS connects to the database and issues three SELECT statements against the table EMP and then disconnects from the database.

In this case, the audit trail 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, and three SELECT statements against the table EMP, and then disconnects from the database.

In this case, the audit trail will contain two records (one for each schema object against which the user issued a SELECT statement in a session).

> **Note:** If you use the BY SESSION option when directing audit records to the operating system audit trail, Oracle generates and stores an audit record each time an access is made. Therefore, in this auditing configuration, BY SESSION is equivalent to BY ACCESS.

BY ACCESS

Setting audit BY ACCESS inserts one audit record into the audit trail for each execution of an auditable operation within a cursor. Events that cause cursors to be reused include the following:

- an application, such as Oracle Forms, holding a cursor open for reuse
- subsequent execution of a cursor using new bind variables
- statements executed within PL/SQL loops where the PL/SQL engine optimizes the statements to reuse a single cursor

Note that auditing is NOT affected by whether a cursor is shared; each user creates her or his own audit trail records on first execution of the cursor.

Example Assume that

- The SELECT TABLE statement auditing option is set BY ACCESS.
- JWARD connects to the database and issues five SELECT statements against the table named DEPT and then disconnects from the database.
- SWILLIAMS connects to the database and issues three SELECT statements against the table DEPT and then disconnects from the database.

The single audit trail contains eight records for the eight SELECT statements.

Defaults and Excluded Operations

The AUDIT command allows you to specify either BY SESSION or BY ACCESS. However, several audit options can be set only BY ACCESS, including

- all statement audit options that audit DDL statements
- all privilege audit options that audit DDL statements

For all other audit options, BY SESSION is used by default.

Auditing By User

Statement and privilege audit options can audit statements issued by any user or statements issued by a specific list of users. By focusing on specific users, you can minimize the number of audit records generated.

> **Additional Information:** See *Oracle8 SQL Reference* for more information about auditing by user.

Example To audit statements by the users SCOTT and BLAKE that query or update a table or view, issue the following statements:

```
AUDIT SELECT TABLE, UPDATE TABLE
     BY scott, blake;
```

28

Database Recovery

These unhappy times call for the building of plans...

Franklin Delano Roosevelt

This chapter introduces the structures that are used during database recovery and describes the Recovery Manager utility, which simplifies backup and recovery operations. The topics in this chapter include:

- An Introduction to Database Recovery
- Structures Used for Database Recovery
- Rolling Forward and Rolling Back
- Recovery Manager
- Performing Recovery in Parallel
- Database Archiving Modes
- Control Files
- Database Backups
- Survivability

> **Additional Information:** The procedures necessary to create and maintain the backup and recovery structures are discussed in the *Oracle8 Backup and Recovery Guide*.

An Introduction to Database Recovery

A major responsibility of the database administrator is to prepare for the possibility of hardware, software, network, process, or system failure. If such a failure affects the operation of a database system, you must usually recover the 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.

Note: The Recovery Manager is a utility that simplifies backup and recovery operations. See "Recovery Manager" on page 28-10.

Additional Information: See the *Oracle8 Backup and Recovery Guide* for detailed information on Recovery Manager and a description of how to recover from loss of data.

Errors and Failures

Several problems can halt the normal operation of an Oracle database or affect database I/O to disk. The following sections describe the most common types. For some of these problems, recovery is automatic and requires little or no action on the part of the database user or database administrator.

User Error

A database administrator can do little to prevent user errors (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 continue.

The Oracle background process PMON detects aborted Oracle processes. If the aborted process is a user or server process, PMON resolves the failure by rolling back the current transaction of the aborted process and releasing any resources that this process was using. Recovery of the failed user or server process is automatic. If the aborted process is a background process, the instance usually cannot continue to function correctly. Therefore, you must shut down and restart the instance.

Network Failure

When your system uses networks (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 30, “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. Instance failure also results when you issue a SHUTDOWN ABORT or STARTUP FORCE statement.

Recovery from Instance Failure 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, recovery from instance failure is relatively automatic. For example, when using the Oracle Parallel Server, other instances perform instance recovery. In single-instance configurations, Oracle performs instance recovery for a database when the database is restarted (mounted and opened to a new instance). The transition from a mounted state to an open state automatically triggers instance recovery, if necessary.

Instance recovery consists of the following steps:

1. Rolling forward to recover data that has not been recorded in the datafiles, yet has been recorded in the online redo log, including the contents of rollback segments.
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 PARTLY 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.

Incremental Checkpointing Incremental checkpointing improves the performance of crash and instance recovery (but *not* media recovery). An incremental checkpoint records the position in the redo thread (log) from which crash or instance recovery needs to begin. This log position is determined by the oldest dirty buffer in the buffer cache. The incremental checkpoint information is maintained periodically with minimal or no overhead during normal processing.

Recovery performance is roughly proportional to the number of buffers that had not been written to the database prior to the crash. You can influence the performance of crash or instance recovery by setting the initialization parameter DB_BLOCK_MAX_DIRTY_TARGET, which specifies an upper bound on the number of dirty buffers that can be present in the buffer cache of an instance at any moment in time. Thus, it is possible to influence recovery time for situations where the buffer cache is very large or where there are stringent limitations on the duration of crash/instance recovery. Smaller values of this parameter impose higher overhead during normal processing since more buffers have to be written. On the other hand, the smaller the value of this parameter, the better the recovery performance, since fewer blocks need to be recovered.

Incremental checkpoint information is maintained automatically by the Oracle server without affecting other checkpoints (such as log switch checkpoints and user-specified checkpoints). In other words, incremental checkpointing occurs independently of other checkpoints occurring in the instance.

Read-Only Tablespaces and Instance Recovery Recovery is not needed on read-only datafiles during instance recovery. Recovery during startup verifies that each online read-only file does not need any media recovery. That is, the file was not restored from a backup taken before it was made read-only. If you restore a read-only tablespace from a backup taken before the tablespace was made read-only, you cannot access the tablespace until you complete media 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 to files on the storage medium.

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.

> **Additional Information:** See the *Oracle8 Backup and Recovery Guide* 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 is archiving the filled online redo log files, Oracle returns an error in the DBW*n* trace file and takes the datafile offline automatically. Only the datafile that cannot be written to is taken offline; the tablespace containing that file remains online.

If the datafile that cannot be written to is in the SYSTEM tablespace, 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 the filled online redo log files, the DBW*n* background process 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. This section introduces each of these structures and its role in database recovery.

Database Backups

A database backup consists of backups of the physical files (all datafiles and a control file) that constitute an Oracle database. To begin 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.

Additional Information: See *Oracle8 Backup and Recovery Guide*.

The Redo Log

The redo log, present for every Oracle database, records all changes made in an Oracle database. The redo log of a database consists of at least two redo log files that are separate from the datafiles (which actually store a database's data). As part of database recovery from an instance or media failure, Oracle applies the appropriate changes in the database's redo log to the datafiles, which updates database data to the instant that the failure occurred.

A database's redo log can consist of two parts: the online redo log and the archived redo log.

The Online Redo Log

Every Oracle database has an associated online redo log. The 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.

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 "Database Archiving Modes" on page 28-16 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.

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 28-19 for more information.

Rolling Forward and Rolling Back

Database buffers in the SGA are written to disk only when necessary, using a least-recently-used algorithm. Because of the way that the DBW*n* process uses this algorithm to write database buffers to datafiles, datafiles might contain some data blocks modified by uncommitted transactions and some data blocks missing changes from committed transactions.

Two potential problems can result if an instance failure occurs:

- Data blocks modified by a transaction might not be written to the datafiles at commit time and might only appear in the redo log. Therefore, the redo log contains changes that must be reapplied to the database during recovery.
- 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.

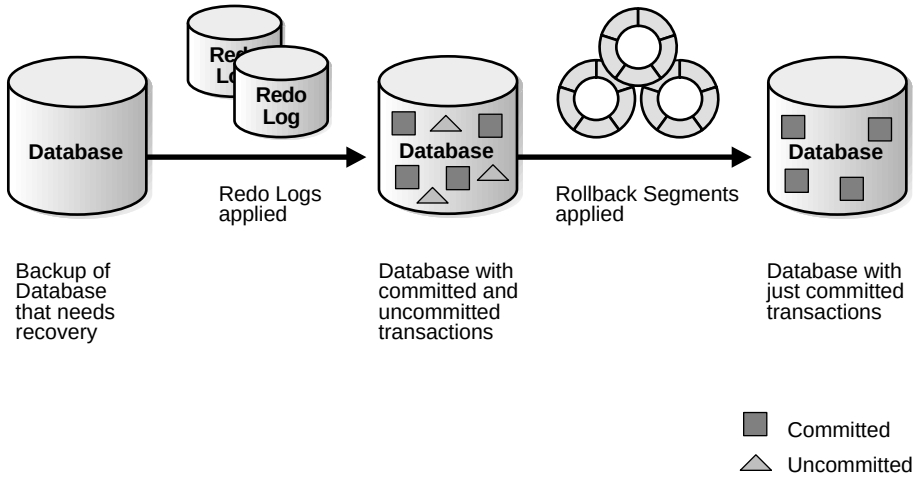
Rollback Segments and Rolling Back

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 28–1 illustrates rolling forward and rolling back, the two steps necessary to recover from any type of system failure.

Figure 28–1 Basic Recovery Steps: Rolling Forward and Rolling Back



Oracle can roll back multiple transactions simultaneously as needed. All transactions system-wide that were active at the time of failure are marked as DEAD. Instead of waiting for SMON to roll back dead transactions, new transactions can recover blocking transactions themselves to get the row locks they need. This feature is called fast transaction rollback.

Recovery Manager

Recovery Manager is a utility that *manages* the processes of creating backups of database files and restoring or recovering files from backups.

> **Additional Information:** See the *Oracle8 Backup and Recovery Guide* for a full description of Recovery Manager.

Recovery Catalog

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

The recovery catalog contains:

- information about backups of datafiles and archivelogs
- information about datafile copies
- information about archived redo logs and copies of them
- information about the physical schema of the target database
- named sequences of commands called *stored scripts*.

The recovery catalog is maintained solely by Recovery Manager. The database server never accesses the recovery catalog directly. Recovery Manager propagates information about backup datafile sets, archived redo logs, and datafile copies into the recovery catalog for long-term retention.

When doing a restore, Recovery Manager extracts the appropriate information from the recovery catalog and passes it to the database server. The server performs various integrity checks on the input files specified for a restore. Incorrect behavior by Recovery Manager cannot corrupt the database.

The Recovery Catalog Database

The recovery catalog is stored in an Oracle database. It is the database administrator's responsibility to make such a database available to Recovery Manager. Taking backups of the recovery catalog is also the database administrator's responsibility. Since the recovery catalog is stored in an Oracle database, you can use Recovery Manager to back it up.

If the recovery catalog is destroyed and no backups are available, then it can be partially reconstructed from the current control file or control file backups.

Operation Without a Recovery Catalog

Use of a recovery catalog is not required. Since most information in the recovery catalog is also available from the control file, Recovery Manager supports an operational mode where it uses only the control file.

This operational mode is appropriate for small databases where installation and administration of another database to serve as the recovery catalog would be burdensome.

Tablespace point-in-time recovery is not supported in this operational mode.

Parallelization

Recovery Manager can parallelize its operations, establishing multiple logon sessions and conducting multiple operations in parallel by using non-blocking UPI. Concurrent operations must operate on disjoint sets of database files.

Attention: Oracle8 can only allocate one Recovery Manager channel at a time, thus limiting the parallelism to one stream. The Oracle8 Enterprise Edition allows unlimited parallelism. See *Getting to Know Oracle8 and the Oracle8 Enterprise Edition* for more information about the features available with Oracle8 and Oracle8 Enterprise Edition.

Parallelization of the **backup**, **copy**, and **restore** commands is handled internally by the Recovery Manager. You only need to specify:

- a list of one or more sequential I/O devices
- the objects to be backed-up, copied, or restored.

Recovery Manager executes commands serially, that is, it completes the previous command before starting the next command. Parallelism is exploited only within the context of a single command. Thus, if 10 datafile copies are desired, it is better to issue a single **copy** command that specifies all 10 copies rather than 10 separate **copy** commands.

Report Generation

The **report** and **list** commands provide information about backups and image copies. The output from these commands is written to the message log file.

The **report** command produces reports that can answer questions such as:

- what files need a backup?
- what files haven't had a backup in a while?
- what backup files can be deleted?

You can use the **report need backup** and **report unrecoverable** commands on a regular basis to ensure that the necessary backups are available to perform recovery, and that the recovery can be performed within a reasonable length of time. The **report deletable** command lists backup sets and datafile copies that can be deleted either because they are redundant or because they could never be used by a **recover** command.

A datafile is considered *unrecoverable* if:

- the only existing backups of the datafile are useless because the archivelogs needed to recover them are not in the recovery catalog or the recovery catalog does not know what logs would be needed
- an unlogged operation has been performed against a schema object residing in the datafile.

(A datafile that does not have a backup is not considered unrecoverable. Such datafiles can be recovered through the use of the **create datafile** command, provided that logs starting from when the file was created still exist.)

The **list** command queries the recovery catalog and produces a listing of its contents. You can use it to find out what backups or copies are available:

- backups or copies of a specified list of datafiles
- backups or copies of any datafile that is a member of a specified list of tablespaces
- backups or copies of any archivelogs with a specified name and/or within a specified range
- incarnations of a specified database.

Performing Recovery in Parallel

Recovery reapplies the changes generated by several concurrent processes, and therefore instance or media recovery can take longer than the time it took to initially generate the changes to a database. With serial recovery, a single process applies the changes in the redo log files sequentially. Using parallel recovery, several processes simultaneously apply changes from redo log files.

Attention: Oracle8 provides limited parallelism with Recovery Manager; the Oracle8 Enterprise Edition allows unlimited parallelism. See *Getting to Know Oracle8 and the Oracle8 Enterprise Edition* for more information about the features available in Oracle8 and Oracle8 Enterprise Edition.

Parallel recovery can be performed manually by spawning several Oracle Enterprise Manager sessions and issuing the RECOVER DATAFILE command on a different set of datafiles in each session. However, this method causes each Oracle Enterprise Manager session to read the entire redo log file.

Instance and media recovery can be parallelized automatically by specifying an initialization parameter or options to the RECOVER command. Oracle uses one process to read the log files sequentially and dispatch redo information to several recovery processes, which apply the changes from the log files to the datafiles. The recovery processes are started automatically by Oracle, so there is no need to use more than one session to perform recovery.

Situations That Benefit from Parallel Recovery

In general, parallel recovery is most effective at reducing recovery time when several datafiles on several different disks are being recovered concurrently. Crash recovery (recovery after instance failure) and media recovery of many datafiles on many different disk drives are good candidates for parallel recovery.

The performance improvement from parallel recovery is also dependent upon whether the operating system supports asynchronous I/O. If asynchronous I/O is not supported, parallel recovery can dramatically reduce recovery time. If asynchronous I/O is supported, the recovery time may only be slightly reduced by using parallel recovery.

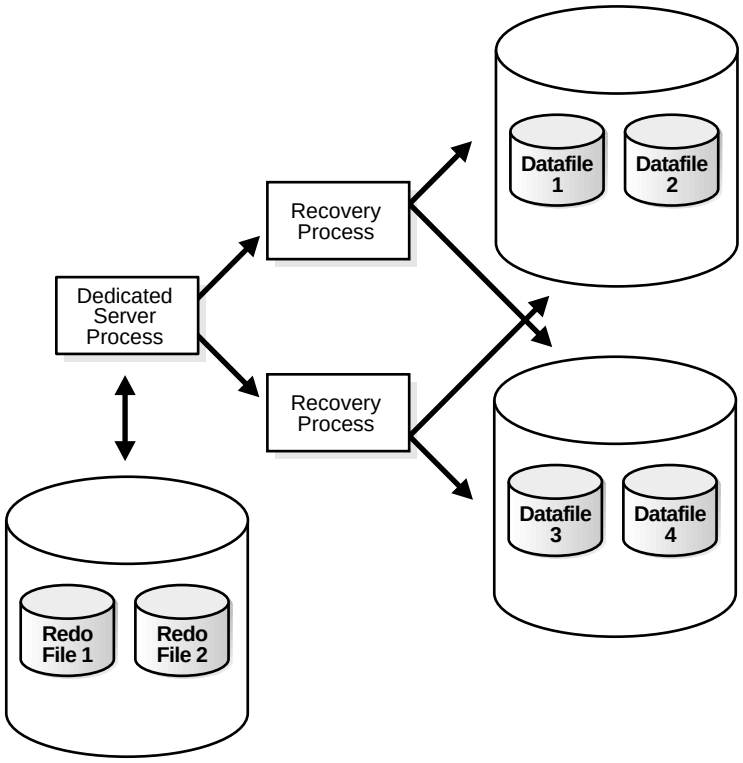
> **Additional Information:** See your operating system documentation to determine whether the system supports asynchronous I/O.

Recovery Processes

In a typical parallel recovery situation, one process is responsible for reading and dispatching redo entries from the redo log files. This is the dedicated server process that begins the recovery session. The server process reading the redo log files enlists two or more recovery processes to apply the changes from the redo entries to the datafiles.

Figure 28–2 illustrates a typical parallel recovery session.

Figure 28–2 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.

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 the LGWR process.

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.

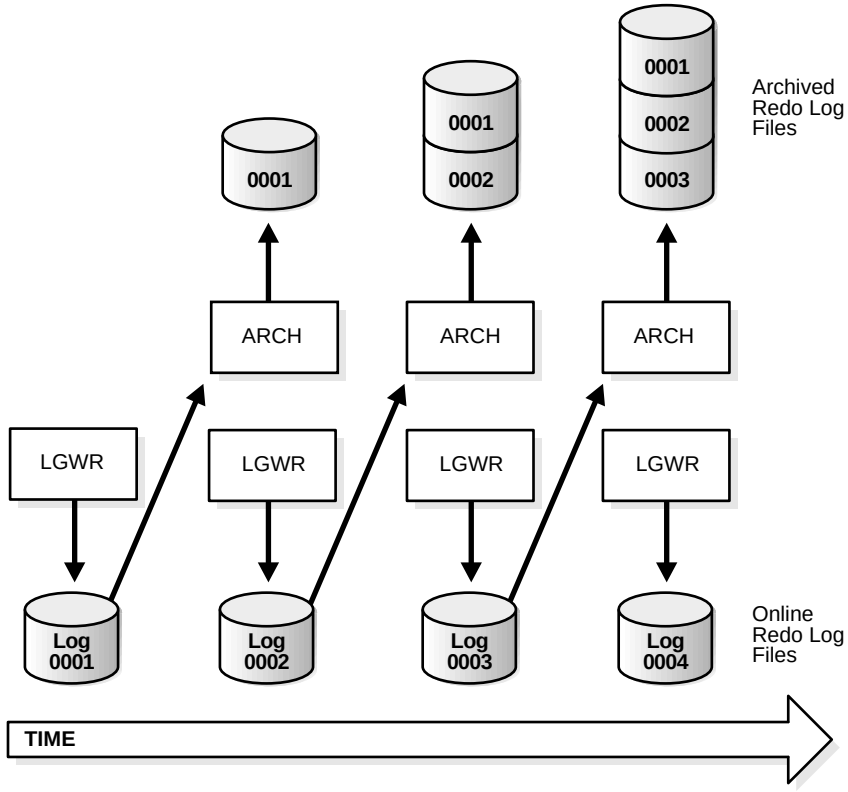
ARCHIVELOG Mode (Media Recovery Enabled)

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 28–3 illustrates how the database's online redo log files are used in ARCHIVELOG mode and how the archived redo log is generated by the process archiving the filled groups (for example, ARCH in this illustration).

ARCHIVELOG mode permits complete recovery from disk failure as well as instance failure, because all changes made to the database are permanently saved in an archived redo log.

Figure 28–3 Online Redo Log File Use in ARCHIVELOG Mode



Automatic Archiving and the ARCH (Archiver) Background Process

An instance can be configured to have an additional background process, the 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, archive one group to one offline storage device, the next group to a different offline storage device, and so on)
- re-archive a group if the original archived version is lost or damaged

When a group is archived manually, the user process issuing the statement to archive a group actually performs the process of archiving the group. Even if the 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 datafiles and online redo log files
- tablespace information
- datafile offline ranges
- the log history
- archived log information
- backup set and backup piece information
- backup datafile and redo log information
- datafile copy information
- the current log sequence number
- checkpoint information

The database name and timestamp originate at database creation. The database's name is taken from either the name specified by the initialization parameter DB_NAME or the name used in the CREATE DATABASE statement.

Each time that a datafile or an online redo log file is added to, renamed in, or dropped from the database, the control file is updated to reflect this physical structure change. These changes are recorded so that

- Oracle can identify the datafiles and online redo log files to open during database startup.
- Oracle can identify files that are required or available in case database recovery is necessary.

Therefore, if you make a change to your database's physical structure, you should immediately make a backup of your control file.

> **Additional Information:** See *Oracle8 Backup and Recovery Guide* 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 about 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.

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.

Database Backups

No matter what backup and recovery scheme you devise for an Oracle database, backups of the database's datafiles and control files are absolutely necessary as part of the strategy to safeguard against potential media failures that can damage these files.

The following sections provide a conceptual overview of the different types of backups that can be made and their usefulness in different recovery schemes.

> **Additional Information:** The *Oracle8 Backup and Recovery Guide* provides more details, along with guidelines for performing database backups.

Whole Database Backups

A *whole database backup* is an operating system backup of all datafiles and the control file that constitute an Oracle database. You can take a whole database backup when the database is shut down or while the database is open. You should not normally take a whole backup after an instance failure or other unusual circumstances.

Consistent Whole Backups vs. Inconsistent Whole Backups

Following a clean shutdown, all of the files that constitute a database are closed and consistent with respect to the current point in time. Thus, a whole backup taken after a shutdown can be used to recover to the point in time of the last whole backup. A whole backup taken while the database is open is not consistent to a given point in time and must be recovered (with the online and archived redo log files) before the database can become available.

Backups and Archiving Mode

The datafiles obtained from a whole backup are useful in any type of media recovery scheme:

- If a database is operating in NOARCHIVELOG mode and a disk failure damages some or all of the files that constitute the database, the most recent consistent whole backup can be used to *restore* (not recover) the database.

  Because an archived redo log is not available to bring the database up to the current point in time, all database work performed since the backup must be repeated. Under special circumstances, a disk failure in NOARCHIVELOG mode can be fully recovered, but you should not rely on this.

- If a database is operating in ARCHIVELOG mode and a disk failure damages

some or all of the files that constitute the database, the datafiles collected by the most recent whole backup can be used as part of database *recovery*.

After restoring the necessary datafiles from the whole backup, database recovery can continue by applying archived and current online redo log files to bring the restored datafiles up to the current point in time.

In summary, if a database is operated in NOARCHIVELOG mode, a consistent whole database backup is the only method to partially protect the database against a disk failure; if a database is operating in ARCHIVELOG mode, either a consistent or an inconsistent whole database backup can be used to restore damaged files as part of database recovery from a disk failure.

Partial Database Backups

A *partial database backup* is any backup short of a whole backup, taken while the database is open or shut down. The following are all examples of partial database backups:

- a backup of all datafiles for an individual tablespace
- a backup of a single datafile
- a backup of a control file

Partial backups are only useful for a database operating in ARCHIVELOG mode. Because an archived redo log is present, the datafiles restored from a partial backup can be made consistent with the rest of the database during recovery procedures.

Datafile Backups

A partial backup includes only some of the datafiles of a database. Individual or collections of specific datafiles can be backed up independently of the other datafiles, online redo log files, and control files of a database. You can back up a datafile while it is offline or online.

Choosing whether to take online or offline datafile backups depends only on the availability requirements of the data — online datafile backups are the only choice if the data being backed up must always be available.

Control File Backups

Another form of a partial backup is a control file backup. Because a control file keeps track of the associated database's physical file structure, a backup of a database's control file should be made every time a structural change is made to the database.

Note: The Recovery Manager automatically backs up the control file in any backup that includes datafile 1, which contains the data dictionary.

Multiplexed control files safeguard against the loss of a single control file. However, if a disk failure damages the datafiles and incomplete recovery is desired, or a point-in-time recovery is desired, a backup of the control file that corresponds to the intended database structure should be used, not necessarily the current control file. Therefore, the use of multiplexed control files is not a substitute for control file backups taken every time the structure of a database is altered.

If you use Recovery Manager to restore the control file prior to incomplete or point-in-time recovery, Recovery Manager automatically restores the most suitable backup control file.

The Export and Import Utilities

Export and Import are utilities used to move Oracle data in and out of Oracle databases. Export is a utility that writes data from an Oracle database to operating system files in an Oracle database format. Export files store information about schema objects created for a database. Import is a utility that reads Export files and restores the corresponding information into an existing database. Although Export and Import are designed for moving Oracle data, they can be used also as a supplemental method of protecting data in an Oracle database.

Additional Information: See *Oracle8 Utilities.*

Read-Only Tablespaces and Backup

You can create backups of a read-only tablespace while the database is open. Immediately after making a tablespace read-only, you should back up the tablespace. As long as the tablespace remains read-only, there is no need to perform any further backups of it.

After you change a read-only tablespace to a read-write tablespace, you need to resume your normal backups of the tablespace, just as you do when you bring an offline read-write tablespace back online.

Bringing the datafiles of a read-only tablespace online does not make these files 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.

Survivability

In the event of a power failure, hardware failure, or any other system-interrupting disaster, Oracle offers the *standby database* feature. The standby database is intended for sites where survivability and disaster recovery are of paramount importance.

Planning for Disaster Recovery

The only way to ensure rapid recovery from a system failure or other disaster is to plan carefully. You must have a set plan with detailed procedures. Whether you are implementing a standby database or you have a single database system, you must have a plan for what to do in the event of a catastrophic failure.

Standby Database

Oracle 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. Oracle provides 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 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 from the primary database.

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, tornado, or earthquake.

> **Additional Information:** See the *Oracle8 Backup and Recovery Guide* for information about creating and maintaining a standby database.

Part VIII

Distributed Processing and Distributed Databases

Part VIII describes distributed processing environments for the Oracle server and database applications, and explains distributed database architecture and data replication across networks.

Part VIII contains the following chapters:

- Chapter 29, “Distributed Processing”
- Chapter 30, “Distributed Databases”
- Chapter 31, “Database Replication”

29

Distributed Processing

We must try to trust one another. Stay and cooperate.

Jomo Kenyatta

This chapter defines distributed processing and describes how the Oracle server and database applications work in a distributed processing environment. This material applies to almost every type of Oracle database system environment.

This chapter includes:

- Oracle Client/Server Architecture
- Distributed Processing

Oracle Client/Server Architecture

In the Oracle database system environment, the database application and the database are separated into two parts: a front-end or *client* portion, and a back-end or *server* portion — hence the term *client/server architecture*. The client executes the database application that accesses database information and interacts with a user through the keyboard, screen, and pointing device such as a mouse. The server executes the Oracle software and handles the functions required for concurrent, shared data access to an Oracle database.

Although the client application and Oracle can be executed on the same computer, greater efficiency can often be achieved when the client portion(s) and server portion are executed by different computers connected via a network. The following sections discuss possible variations in the Oracle client / server architecture.

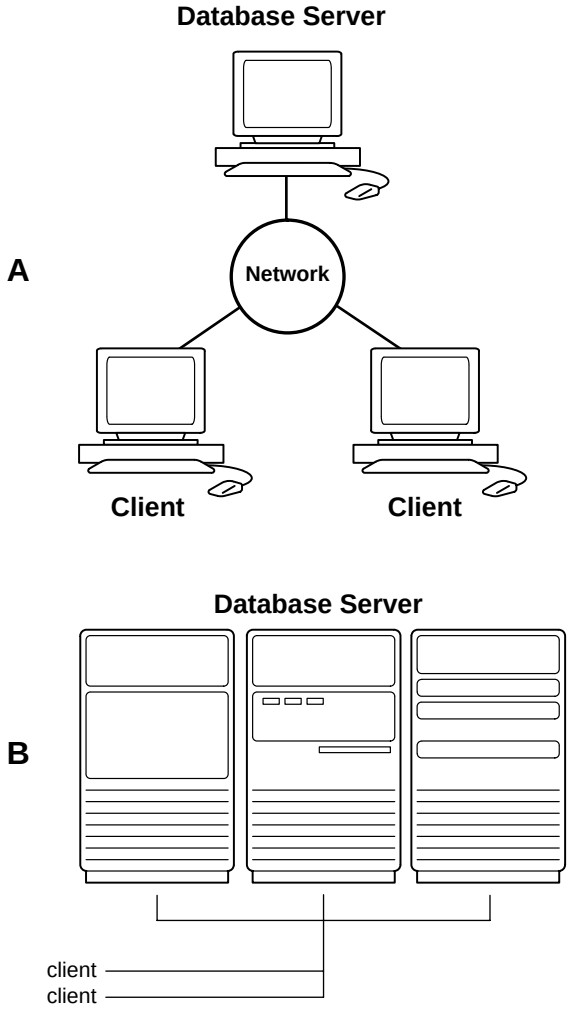
Distributed Processing

Distributed processing is the use of more than one processor to perform the processing for an individual task. Examples of distributed processing in Oracle database systems appear in Figure 29–1.

- In Part A of the figure, the client and server are located on different computers; these computers are connected via a network.
- In Part B of the figure, a single computer has more than one processor, and different processors separate the execution of the client application from Oracle.

Note: This chapter applies to environments with one database on one server. In a *distributed database*, one server (Oracle) may need to access a database on another server. See Chapter 30, “Distributed Databases”, for more information about clients and servers in distributed databases.

Figure 29–1 The Client/Server Architecture and Distributed Processing



In the networked example (Part A), the server and clients communicate via Net8, Oracle's network interface. See "Net8" on page 29-5 for more information.

Oracle client / server architecture in a distributed processing environment provides the following benefits:

- Client applications are not responsible for performing any data processing. Rather, they request input from users, request data from the server, and then analyze and present this data using the display capabilities of the client workstation or the terminal (for example, using graphics or spreadsheets).
- Client applications are not dependent on the physical location of the data. If the data is moved or distributed to other database servers, the application continues to function with little or no modification.
- Oracle exploits the multitasking and shared-memory facilities of its underlying operating system. As a result, it delivers the highest possible degree of concurrency, data integrity, and performance to its client applications.
- Client workstations or terminals can be optimized for the presentation of data (for example, by providing graphics and mouse support) and the server can be optimized for the processing and storage of data (for example, by having large amounts of memory and disk space).
- In networked environments, you can use inexpensive client workstations to access the remote data of the server effectively.
- If necessary, Oracle can be *scaled a*s your system grows. You can add multiple servers to distribute the database processing load throughout the network (*horizontally scaled*), or you can move Oracle to a minicomputer or mainframe, to take advantage of a larger system's performance (*vertically scaled*). In either case, all data and applications are maintained with little or no modification, since Oracle is portable between systems.
- In networked environments, shared data is stored on the servers, rather than on all computers in the system. This makes it easier and more efficient to manage concurrent access.
- In networked environments, client applications submit database requests to the server using SQL statements. Once received, the SQL statement is processed by the server, and the results are returned to the client application. Network traffic is kept to a minimum because only the requests and the results are shipped over the network.

Net8

Net8 is the Oracle network interface that allows Oracle tools running on network workstations and servers to access, modify, share, and store data on other servers. Net8 is considered part of the program interface in network communications. See Chapter 7, “Process Structure”, for more information about the program interface.

Net8 uses the communication protocols or application programmatic interfaces (APIs) supported by a wide range of networks to provide a distributed database and distributed processing for Oracle.

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

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

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

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

How Net8 Works

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

The Net8 drivers take SQL statements from the interface of the Oracle tools and package them for transmission to Oracle via one of the supported industry-standard higher level protocols or programmatic interfaces. The drivers also take replies from Oracle and package them for transmission to the tools via the same

higher level communications mechanism. This is all done independently of the network operating system.

Additional Information: Depending on the operating system that executes Oracle, the Net8 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.

Refer to the *Oracle Net8 Administrator's Guide* for additional information on Net8.

30

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.

René Descartes: *Le Discours de la Methode*

This chapter describes the basic concepts and terminology of Oracle's distributed database architecture. The chapter includes:

- Oracle's Distributed Database Architecture
- Heterogeneous Distributed Databases
- Developing Distributed Database Applications
- Administering an Oracle Distributed Database System
- National Language Support

Oracle's Distributed Database Architecture

A *distributed database* is a set of databases stored on multiple computers that typically appears to applications as a single database. Consequently, an application can simultaneously access and modify the data in several databases in a network. Each Oracle database in the system is controlled by its local Oracle server but cooperates to maintain the consistency of the global distributed database.

Figure 30–1 illustrates a representative Oracle distributed database system.

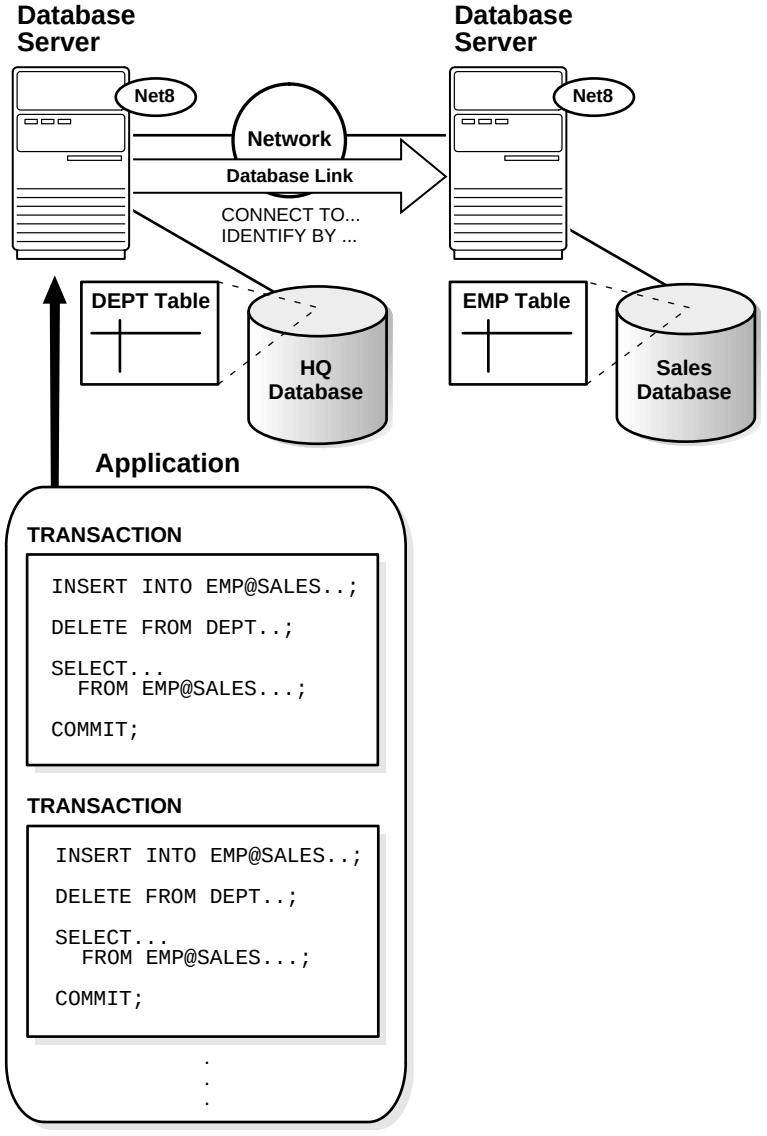
Clients and Servers

A database *server* is the Oracle software managing a database, and a *client* is an application that requests information from a server. Each computer in a system is a *node*. A node in a distributed database system act as a client, a server, or both, depending on the situation. For example, in Figure 30–1, the computer that manages the HQ database is acting as a database server when a statement is issued against its local data (for example, the second statement in each transaction issues a 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).

Direct and Indirect Connections

A client can connect directly or indirectly to a database server. In Figure 30–1, 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 30–1 An Oracle Distributed Database System



The Network

To link the individual databases of a distributed database system, a network is necessary. The following sections explain more about network issues in an Oracle distributed database system.

Net8

All Oracle databases in a distributed database system use Oracle's networking software, Net8, to facilitate inter-database communication across a network. Just as Net8 connects clients and servers that operate on different computers of a network, it also allows database servers to communicate across networks to support remote and distributed transactions in a distributed database.

Net8 makes transparent the connectivity that is necessary to transmit SQL requests and receive data for applications that use the system. Net8 takes SQL statements from a client and packages them for transmission to an Oracle server over a supported industry-standard communication protocol or programmatic interfaces. Net8 also takes replies from a server and packages them for transmission back to the appropriate client. Net8 performs all processing independent of an underlying network operating system.

> **Additional Information:** See the *Oracle Net8 Administrator's Guide* for more information about Net8 and its features.

Oracle Names

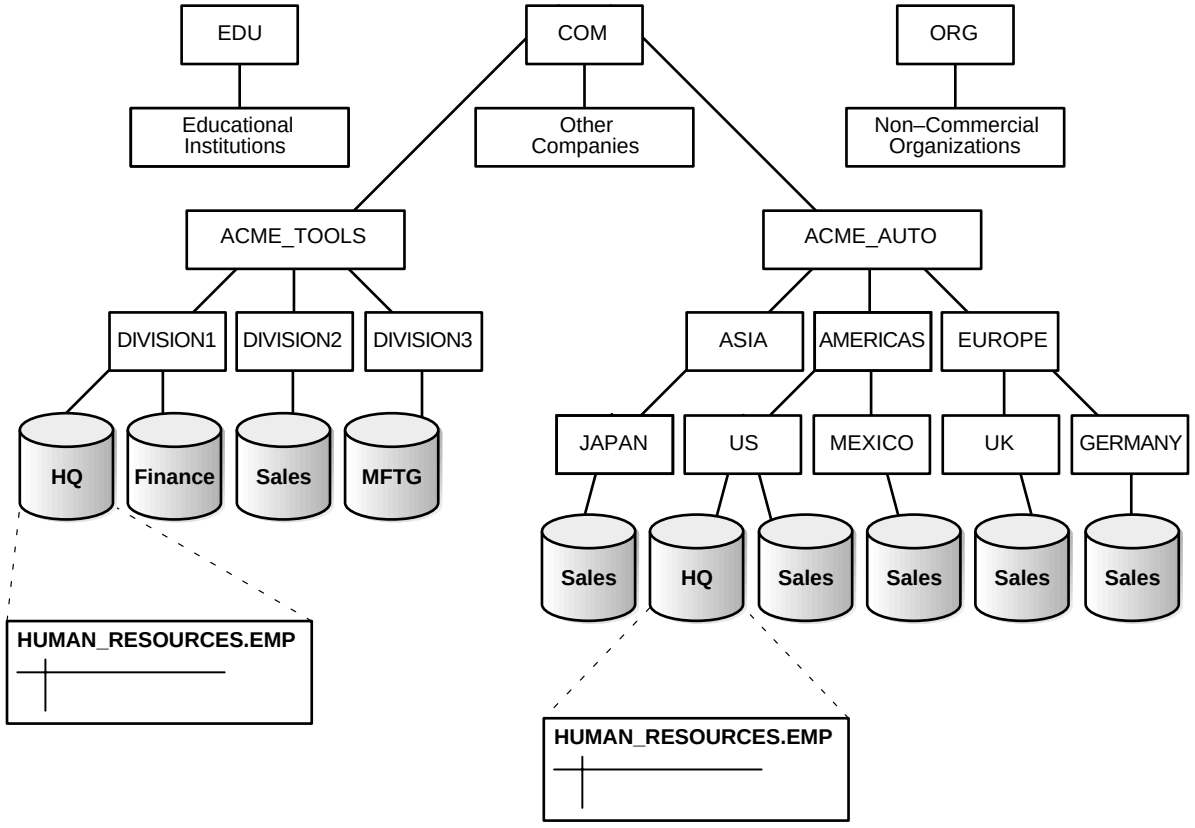
Optionally, an Oracle network can use Oracle Names to provide the system with a global directory service. When an Oracle network supports a distributed database system, you can use Oracle Names servers as a central repositories of information about each database in the system to ease the configuration of distributed database access.

Databases and Database Links

Each database in a distributed database is distinct from all other databases in the system and has its own *global database name*. Oracle forms a database's global database name by prefixing the database's network domain with the individual database's name.

For example, Figure 30–2 illustrates a representative hierarchical arrangement of databases throughout a network.

Figure 30–2 Network Directories and Global Database Names



While several databases can have the same individual name, each database must have a unique global database name.

For example, the network domains US.AMERICAS.ACME_AUTO.COM and UK.EUROPE.ACME_AUTO.COM in Figure 30–2 each contain a SALES database:

```
SALES.US.AMERICAS.ACME_AUTO.COM

SALES.UK.EUROPE.ACME_AUTO.COM
```

Database Links

To facilitate application requests in a distributed database system, Oracle uses *database links*. A database link defines a one-way communication path from an Oracle database to another database.

Database links are essentially transparent to the users of an Oracle distributed database system, 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 SQL statement creates a database link in the local database that describes a path to the remote SALES.US.AMERICAS.ACME_AUTO.COM database.

```
CREATE DATABASE LINK sales.us.americas.acme_auto.com ... ;
```

After creating a database link, applications connected to the local database can access data in the remote SALES.US.AMERICAS.ACME_AUTO.COM database. The next section explains how applications can reference remote schema objects in a distributed database and includes examples of how SQL statements use database links.

> **Additional Information:** Oracle supports several different types of database links. See *Oracle8 Distributed Database Systems* for more information.

Schema Object Name Resolution

To resolve application references to schema objects (a process called *name resolution*) Oracle forms object names using a hierarchical approach. For example, within a single database, Oracle guarantees that each schema has a unique name, and that within a schema, each object has a unique name. As a result, a schema object's name is always unique within the database. Furthermore, Oracle can easily resolve application references to a schema object's local name.

In a distributed database, a schema object such as a table is accessible to all applications in the system. Oracle simply extends the hierarchical naming model with global database names to effectively create *global object names* and resolve references to the schema objects in a distributed database system. For example, a query can reference a remote table by specifying its fully qualified name, including the database in which it resides.

```
SELECT * FROM scott.emp@sales.us.americas.acme_auto.com;
```

To complete the request, the local database server implicitly uses a database link that connects to the remote SALES database.

Connecting Between Oracle Server Versions

An Oracle distributed database system can incorporate Oracle databases of different versions. All supported releases of Oracle can participate in a distributed database system. However, the applications that work with the distributed database must understand the functionality that is available at each node in the system. For example, a distributed database application cannot expect an Oracle7 database to understand the object SQL extensions that are available with Oracle8.

Distributed Databases and Distributed Processing

The terms "distributed database" and "distributed processing" are closely related, but have very distinct meanings.

Distributed Database	A distributed database is a set of databases stored on multiple computers that appears to applications as a single database.
Distributed Processing	Distributed processing occurs when an application system distributes its tasks among different computers in a network. For example, a database application typically distributes front-end presentation tasks to client PCs or NCs and allows a back-end database server to manage shared access to a database. Consequently, a distributed database application processing system is more commonly referred to as a "client-server" database application system.

Oracle distributed database systems employ a distributed processing architecture to function. For example, an Oracle server acts as a client when it requests data that another Oracle server manages.

Distributed Databases and Database Replication

The terms "distributed database" and "database replication" are also closely related, yet different. In a pure distributed database, the system manages a single copy of all data and supporting database objects. Distributed database applications typically use distributed transactions to access both local and remote data and modify the global database in real-time.

Note: This chapter discusses pure distributed databases. See Chapter 31, "Database Replication" for a discussion of replication.

Replication is the process of copying and maintaining database objects in multiple databases that make up a distributed database system. While replication relies on distributed database technology to function, database replication can offer applications benefits that are not possible within a pure distributed database environment. Most commonly, replication is useful to improve the performance and protect the availability of applications because alternate data access options exist. For example, an application might normally access a local database rather than a remote server to minimize network traffic and achieve maximum performance. Furthermore, the application can continue to function if the local server experiences a failure, but other servers with replicated data remain accessible.

> **Additional Information:** See *Oracle8 Replication* for more information about Oracle's replication features.

Heterogeneous Distributed Databases

A *heterogeneous distributed database* is a distributed database in which at least one of the databases is a non-Oracle system. Access to non-Oracle systems from an Oracle server is provided by *Oracle Open Gateways*. The Oracle server, together with the gateway, can provide full heterogeneity transparency to the application. That is, the application doesn't have to be aware that a non-Oracle system is accessed.

Database links are used between the Oracle server and the non-Oracle system to access the data in the non-Oracle system, or to execute a remote procedure in the non-Oracle system. By integrating the transactional system of the non-Oracle system with the Oracle server, the integrity of the data can be guaranteed.

Transparent SQL Access

The application just issues Oracle SQL statements against the local Oracle server. The SQL statement can refer to data in a remote non-Oracle system, just as if it were a remote Oracle system. The Oracle server, together with the gateway, will perform the necessary translations to access the non-Oracle system in its own SQL dialect.

Procedural Access

Some non-Oracle systems need to be accessed procedurally. The application just issues a PL/SQL remote procedure call, and the Oracle server, together with the gateway, will perform translations to execute procedures or functions in the remote non-Oracle system. For example, the remote procedure could interface with a messaging system and put messages in the non-Oracle messaging system. If the messaging system has transactional support, the Oracle server together with the

gateway, could perform operations in the messaging system in a larger Oracle distributed transaction, guaranteeing that either all changes (both in the messaging system and in the Oracle server) are committed or rolled back.

Gateway Features

In summary, features of Oracle Open Gateways include, but are not limited to:

- *Distributed Transactions.* A transaction can span both Oracle and non-Oracle systems, while still guaranteeing, through Oracle's two phase commit mechanism, that changes are either all committed or all rolled back.
- *Transparent SQL access.* Integrate data from non-Oracle systems into the Oracle environment as if the data is stored in one single, local database. SQL statements issued by the application are transparently transformed into SQL statement understood by the non-Oracle system.
- *Procedural Access.* Procedural systems, like messaging and queuing systems, are accessed from an Oracle server using PL/SQL remote procedure calls.
- *Data Dictionary translations.* To make the non-Oracle system appear as another Oracle server, SQL statements containing references to Oracle's data dictionary tables are transformed into SQL statements containing references to a non-Oracle system's data dictionary tables.
- *Pass-through SQL.* Optionally, application programmers can directly access a non-Oracle system from an Oracle application using the non-Oracle system's SQL dialect.
- *Accessing stored procedures.* Stored procedures in SQL-based non-Oracle systems are accessed as if they were PL/SQL remote procedures.
- *National Language Support.* Gateways supports multibyte character sets, and will translate character sets between a non-Oracle system and the Oracle server.
- *Global query optimization*. Cardinality and indexes on tables at the non-Oracle system are taken into account by the Oracle query optimizer and decomposed to produce efficient SQL statements to be executed at the non-Oracle system.

> **Additional Information:** Not all features listed above might apply to your particular gateway. See your gateway documentation for the supported features.

Version 8 Gateways

Version 8 Gateways are tightly integrated with the Oracle server, by using the Oracle feature Heterogeneous Services. Heterogeneous Services is integrated into the Oracle server, and therefore, administration tasks for heterogeneous access are now generic across the different gateways, and integrated into the Oracle server.

> **Additional Information:** See *Oracle8 Replication* for more information about Heterogeneous Services.

Version 4 Gateways

Version 4 gateways are supported against Oracle7 and Oracle8 servers.

> **Additional Information:** See *Oracle Open Gateway Technology, Concepts and Administration Guide Version 4* for gateway information.

Developing Distributed Database Applications

When you build applications on top of a distributed database system, there are several issues to consider. The following sections explain how applications access data in a distributed database.

Remote and Distributed SQL 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. For example:

```
SELECT * FROM scott.dept@sales.us.americas.acme_auto.com;
```

A *remote update* is an update that modifies data in one or more tables, all of which are located at the same remote node. For example:

```
UPDATE scott.dept@sales.us.americas.acme_auto.com
  SET loc = 'NEW YORK'
  WHERE deptno = 10;
```

> **Note:** A remote update may include a subquery that retrieves data from one or more remote nodes, but because the update happens at only a single remote node, the statement is classified as a remote update.

A *distributed query* retrieves information from two or more nodes. For example:

```
SELECT ename, dname
  FROM scott.emp e, scott.dept@sales.us.americas.acme_auto.com d
  WHERE e.deptno = d.deptno;
```

A *distributed update* modifies data on two or more nodes. A distributed update is possible using a PL/SQL subprogram unit, such as a procedure or trigger, that includes two or more remote updates that access data on different nodes. For example:

```
BEGIN
  UPDATE scott.dept@sales.us.americas.acme_auto.com
    SET loc = 'NEW YORK'
    WHERE deptno = 10;
  UPDATE scott.emp
    SET deptno = 11
    WHERE deptno = 10;
END;
```

Statements in the program are sent to the remote nodes, and the execution of it succeeds or fails as a unit.

Remote Procedure Calls (RPCs)

Developers can code PL/SQL packages and procedures to support applications that work with a distributed database. Applications can make local procedure calls to perform work at the local database and *remote procedure calls (RPCs)* to perform work at a remote database. When a program calls a remote procedure, the local server passes all procedure parameters to the remote server in the call. For example:

```
BEGIN
  emp_mgmt.del_emp@sales.us.americas.acme_auto.com(1257);
END;
```

When developing packages and procedures for distributed database systems, developers must code with an understanding of what program units should do at remote locations, and how to return the results to a calling application.

Remote and Distributed Transactions

A *remote transaction* contains one or more remote statements, all of which reference the same remote node. For example:

```
UPDATE scott.dept@sales.us.americas.acme_auto.com
  SET loc = 'NEW YORK'
  WHERE deptno = 10;
```

```
UPDATE scott.emp@sales.us.americas.acme_auto.com
  SET deptno = 11
  WHERE deptno = 10;
COMMIT;
```

A *distributed transaction* contains one or more statements that, individually or as a group, update data on two or more distinct nodes of a distributed database. For example:

```
UPDATE scott.dept@sales.us.americas.acme_auto.com
  SET loc = 'NEW YORK'
  WHERE deptno = 10;
UPDATE scott.emp
  SET deptno = 11
  WHERE deptno = 10;
COMMIT;
```

Note: If all statements of a transaction reference only a single remote node, the transaction is remote, not distributed.

Two-Phase Commit Mechanism

A DBMS must guarantee that all statements in a transaction, distributed or non-distributed, either commit or rollback as a unit, so that if the transaction is designed properly, the data in the logical database is always consistent. The effects of an ongoing transaction should be 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 15, “Transaction Management”. In a distributed database, Oracle must coordinate transaction control with the same characteristics over a network and maintain data consistency, even if a network or system failure occurs.

Oracle’s *two-phase commit* mechanism guarantees that *all* database servers participating in a distributed transaction either all commit or all roll back the statements in the transaction. A two-phase commit mechanism also protects implicit DML operations performed by integrity constraints, remote procedure calls, and triggers.

Additional Information: *Oracle8 Distributed Database Systems* has more information about Oracle’s two-phase commit mechanism.

Transparency in a Distributed Database System

With minimal effort, you can make the functionality of an Oracle distributed database system transparent to users that work with the system. The goal of transparency is to make a distributed database system appear as though it is a single Oracle database. Consequently, the system does not burden developers and users of the system with complexities that would otherwise make distributed database application development challenging and detract from user productivity.

The following sections explain more about transparency in a distributed database system.

Location Transparency

An Oracle distributed database system has features that allow application developers and administrators to hide the physical location of database objects from applications and users. *Location transparency* exists when a user can universally refer to a database object such as a table, regardless of the node to which an application connects. Location transparency has several benefits, including:

- Access to remote data is simple, because database users do not need to know the physical location of database objects.
- Administrators can move database objects with no impact on end-users or existing database applications.

Most typically, administrators and developers use synonyms to establish location transparency for the tables and supporting objects in an application schema. For example, the following statements create synonyms in a database for tables in another, remote database.

```
CREATE PUBLIC SYNONYM emp
  FOR scott.emp@sales.us.americas.acme_auto.com
CREATE PUBLIC SYNONYM dept
  FOR scott.dept@sales.us.americas.acme_auto.com
```

Now, rather than access the remote tables with a query such as:

```
SELECT ename, dname
  FROM scott.emp@sales.us.americas.acme_auto.com e,
       scott.dept@sales.us.americas.acme_auto.com d
  WHERE e.deptno = d.deptno;
```

an application can issue a much simpler query that does not have to account for the location of the remote tables.

```
SELECT ename, dname
  FROM emp e, dept d
  WHERE e.deptno = d.deptno;
```

In addition to synonyms, developers can also use views and stored procedures to establish location transparency for applications that work in a distributed database system.

Statement and Transaction Transparency

Oracle's distributed database architecture also provides query, update, and transaction transparency. For example, standard SQL commands such as SELECT, INSERT, UPDATE, and DELETE work just as they do in a non-distributed database environment. Additionally, applications control transactions using the standard SQL commands COMMIT, SAVEPOINT, and ROLLBACK — there is no requirement for complex programming or other special operations to provide distributed transaction control.

- The statements in a single transaction can reference any number of local or remote tables.
- Oracle guarantees that all nodes involved in a distributed transaction take the same action: they either all commit or all roll back the transaction.
- If a network or system failure occurs during the commit of a distributed transaction, the transaction is automatically and transparently resolved globally; that is, when the network or system is restored, the nodes either all commit or all roll back the transaction.

Internal Operations Each committed transaction has an associated *system change number* (*SCN*) to uniquely identify the changes made by the statements within that transaction. In a distributed database, the SCNs of communicating nodes are coordinated when:

- A connection is established using the path described by one or more database links.
- A distributed SQL statement is executed.
- A distributed transaction is committed.

Among other benefits, the coordination of SCNs among the nodes of a distributed database system allows global distributed read-consistency at both the statement and transaction level. If necessary, global distributed time-based recovery can also be completed.

Replication Transparency

Oracle also provides many features to transparently replicate data among the nodes of the system.

> **Additional Information:** See *Oracle8 Replication* for more information about Oracle's replication features.

Administering an Oracle Distributed Database System

Just as there are unique issues to consider when developing applications for an Oracle distributed database system, there are special issues to understand for distributed database administration. The following sections explain the some special topics for managing databases in an Oracle distributed database system.

Site Autonomy

Site autonomy means that each server participating in a distributed database is administered independently from all other databases, as though each database operates as a non-distributed database. Although several databases can work together, each database is a distinct, separate repository of data that you manage individually. Some of the benefits of site autonomy in an Oracle distributed database include:

- Nodes of the system can mirror the logical organization of companies or cooperating organizations that need to maintain an "arms length" relationship.
- Local database administrators control corresponding local data. Therefore, each database administrator's domain of responsibility is smaller and more manageable.
- Independent failures are less likely to disrupt other nodes of the distributed database. The global Oracle 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.
- Administrators can recovery from isolated system failures independent of other nodes in the system.
- A data dictionary exists for each local database — a global catalog is not necessary to access local data.
- Nodes can upgrade software independently.

Although Oracle allows you to manage each database in a distributed database system independently, that is not to say that you should ignore the global require-

ments of the system. For example, additional user accounts might be necessary in each database are necessary to support the links that you create to facilitate server-to-server connections. The following sections explain more about these particular topics and demonstrate the need for a global perspective of the entire distributed database environment when managing individual nodes in the system.

Distributed Database Security

Oracle supports all of the security features that are available with a non-distributed database environment for distributed database systems, including:

- password or external service authentication for users and roles
- login packet encryption for client-to-server and server-to-server connections

The following sections explain some additional topics to consider when configuring an Oracle distributed database system.

Supporting User Accounts and Roles

In a distributed database system, you must carefully plan the user accounts and roles that are necessary to support applications using the system.

- The user accounts necessary to establish server-to-server connections must be available in all databases of the distributed database system.
- The roles necessary to make available application privileges to distributed database application users must be present in all databases of the distributed database system.

As you create the database links for the nodes in a distributed database system, determine what user accounts and roles each site needs to support server-to-server connections that use the links.

> **Additional Information:** See *Oracle8 Distributed Database Systems* for more information about the user accounts that must be available to support different types of database links in the system.

Global Users and Roles

In a distributed environment, users typically require access to many network services. When it's necessary to configure separate authentications for each user to access each network service, security administration can become unwieldy, especially for large systems. The use of a global authentication service is a common technique for simplifying security management for distributed environments.

In an Oracle client/server or distributed database environment, you have two options to support global authentication for users and roles:

- Oracle Security Server is a product that supports centralized authentication and distributed authentication in an Oracle network. Oracle Security Server is a standard option of Oracle.
- When global database user and role authentication must work within the framework of a non-Oracle authentication service (for example, DCE), an Oracle distributed database environment can use Net8's Advanced Networking Option. The Net8 Advanced Networking Option is an optional product that bundles a number of features that you can use to enhance Net8 and the security of an Oracle distributed database system.

> **Attention:** The Advanced Networking Option is available only if you have purchased Oracle8 Enterprise Edition.

Data Encryption

The Net8 Advanced Networking Option also enables Net8 and related products to use network data encryption and checksumming so that data cannot be read or altered. It protects data from unauthorized viewing by using the RSA Data Security RC4 or the Data Encryption Standard (DES) encryption algorithm. To ensure that data has not been modified, deleted, or replayed during transmission, the security services of the Advanced Networking Option can generate a cryptographically secure message digest and include it with each packet sent across the network.

> **Additional Information:** See the *Oracle Net8 Administrator's Guide* for more information about these and other features of the Advanced Networking Option. Also see *Getting to Know Oracle8 and the Oracle8 Enterprise Edition* for information about the features and options that are available with Oracle8 Enterprise Edition.

Tools for Administering Oracle Distributed Databases

The database administrator has several choices for tools to use when managing an Oracle distributed database system:

- Oracle Enterprise Manager
- third-party administration tools
- SNMP support

Oracle Enterprise Manager

Oracle Enterprise Manager is Oracle's database administration tool. The graphical component of Oracle Enterprise Manager allows you to perform database administration tasks with the convenience of a graphical user interface (GUI). The line mode component of Oracle Enterprise Manager provides a line-mode interface.

Oracle Enterprise Manager provides administrative functionality via an easy-to-use interface. You can use Oracle Enterprise Manager to:

- Perform traditional administrative tasks, such as database startup, shutdown, backup, and recovery. Rather than manually entering the SQL commands to perform these tasks, you can use Oracle Enterprise Manager's graphical interface to execute the commands quickly and conveniently by pointing and clicking with the mouse.
- Concurrently perform multiple tasks. Because you can open multiple windows simultaneously in Oracle Enterprise Manager, you can perform multiple administrative and non-administrative tasks concurrently.
- Administer multiple databases. You can use Oracle Enterprise Manager to administer a single database or to simultaneously administer multiple databases.
- Centralize database administration tasks. You can administer both local and remote databases running on any Oracle platform in any location worldwide. In addition, these Oracle platforms can be connected by any network protocol(s) supported by Net8.
- Dynamically execute SQL, PL/SQL, and Oracle Enterprise Manager commands. You can use Oracle Enterprise Manager to enter, edit, and execute statements. Oracle Enterprise Manager also maintains a history of statements executed. Thus, you can re-execute statements without retyping them, a particularly useful feature if you need to execute lengthy statements repeatedly in a distributed database system.
- Perform administrative tasks using Oracle Enterprise Manager's line-mode interface when a graphical user interface is unavailable or undesirable.

Third-Party Administration Tools

Currently more than 60 companies produce more than 150 products that help manage Oracle databases and networks, providing a truly open environment.

SNMP Support

Besides its network administration capabilities, Oracle *Simple Network Management Protocol* (*SNMP*) support allows an Oracle server to be located and queried by any SNMP-based network management system. SNMP is the accepted standard underlying many popular network management systems such as:

- HP's OpenView
- Digital's POLYCENTER Manager on NetView
- IBM's NetView / 6000
- Novell's NetWare Management System
- SunSoft's SunNet Manager

> **Additional Information:** See the *Oracle SNMP Support Reference Guide*.

National Language Support

Oracle supports 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.

> **Additional Information:** See *Oracle8 Reference* for more information about National Language Support features.

31

Database Replication

Lady, you are the cruel'st she alive,
If you will lead these graces to the grave
And leave the world no copy.

Shakespeare: *Twelfth-Night*

This chapter explains the basic concepts and terminology related to the Oracle replication features.

- What Is Replication?
- Basic Replication Concepts
- Advanced Replication Concepts

Attention: The Advanced Replication features described in this chapter are available only if you have purchased Oracle8 Enterprise Edition.

Additional Information: *Oracle8 Replication* contains detailed information about database replication.

What Is Replication?

Replication is the process of copying and maintaining database objects in multiple databases that make up a distributed database system. Replication can improve the performance and protect the availability of applications because alternate data access options exist. For example, an application might normally access a local database rather than a remote server to minimize network traffic and achieve maximum performance. Furthermore, the application can continue to function if the local server experiences a failure, but other servers with replicated data remain accessible.

Oracle supports two different forms of replication: basic and advanced replication.

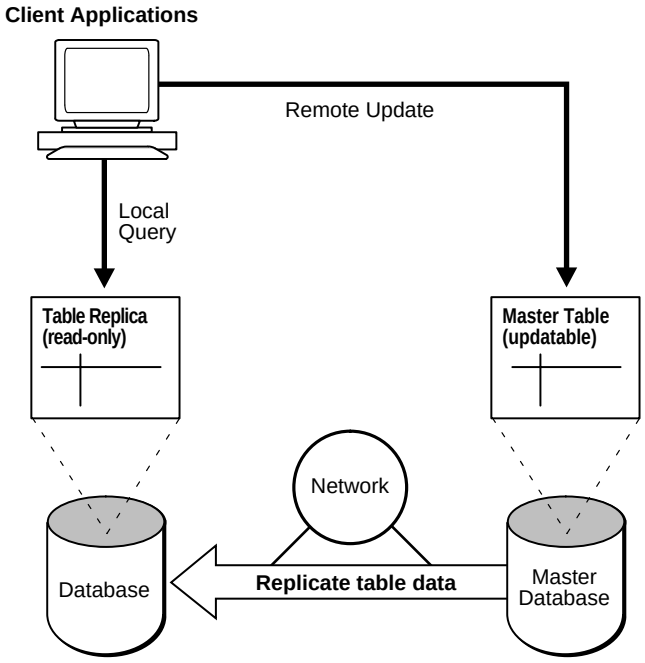
Basic Replication

With *basic replication*, data replicas provide read-only access to the table data that originates from a primary (master) site. Applications can query data from local data replicas to avoid network access regardless of network availability. However, applications throughout the system must access data at the primary site when updates are necessary.

Figure 31–1 illustrates basic replication.

Oracle can support basic, read-only replication environments using *read-only table snapshots*. To learn more about basic replication and read-only snapshots, see “Basic Replication Concepts” on page 31-4.

Figure 31–1 Basic, Read-Only Replication



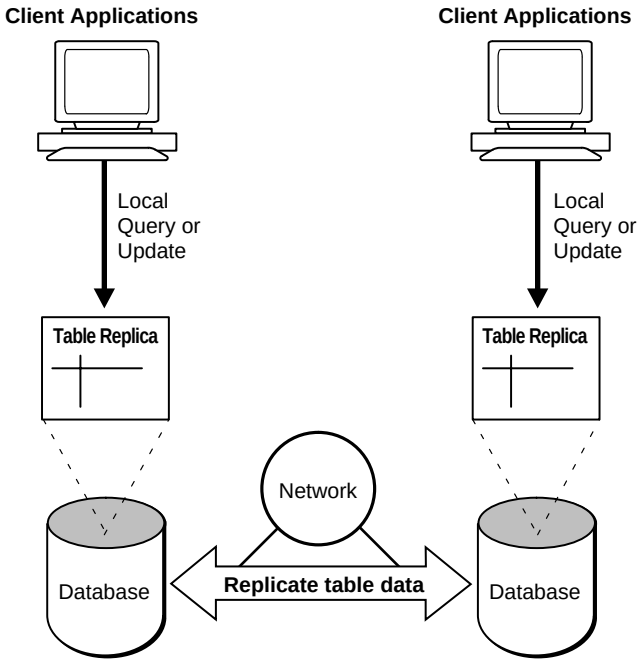
Advanced (Symmetric) Replication

The Oracle *advanced replication* features extend the capabilities of basic read-only replication by allowing applications to update table replicas throughout a replicated database system. With advanced replication, data replicas anywhere in the system can provide both read and update access to a table's data. Participating Oracle database servers automatically work to converge the data of all table replicas, and ensure global transaction consistency and data integrity.

Figure 31–2 illustrates advanced replication.

Oracle can support the requirements of advanced replication environments using several configurations. To learn more about advanced replication systems, see "Advanced Replication Concepts" on page 31-11.

Figure 31–2 Advanced Replication.



Basic Replication Concepts

Basic replication environments support applications that require read-only access to the table data that originates from a primary site. The following sections explain the fundamental concepts of basic replication environments.

- Uses of Basic Replication
- Read-Only Table Snapshots
- Snapshot Refreshes

Uses of Basic Replication

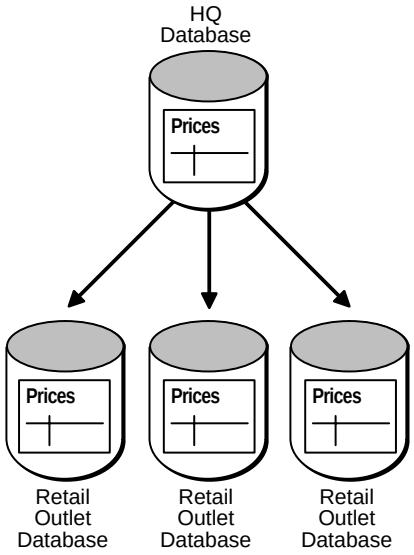
Basic, read-only data replication is useful for several types of applications.

Information Distribution

Basic replication is useful for information distribution. For example, consider the operation of a large consumer department store chain. With this type of business, it

is critical to ensure that product price information is always available, relatively current, and consistent at all retail outlets. To achieve these goals, each retail store can have its own product price data that it refreshes nightly from a primary price table at corporate headquarters.

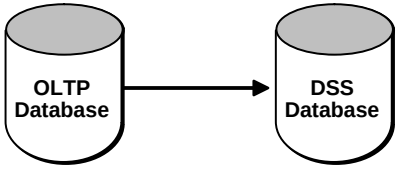
Figure 31–3 Information Distribution



Information Off-Loading

Basic replication is useful as a way to replicate entire databases or off-load information. For example, when the performance of high-volume transaction processing system is critical, it can be advantageous to maintain a duplicate database to isolate the demanding queries of decision support applications.

Figure 31–4 Information Off-Loading



Information Transport

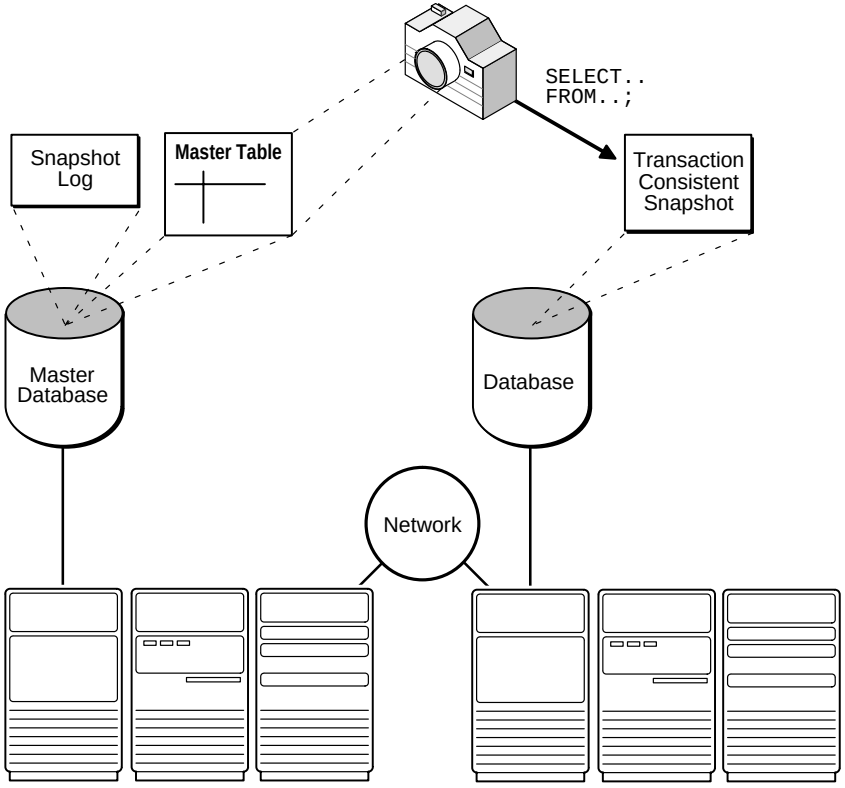
Basic replication can be useful as an information transport mechanism. For example, basic replication can periodically move data from a production transaction processing database to a data warehouse.

Read-Only Table Snapshots

A *read-only table snapshot* is a local copy of table data that originates from one or more remote master tables. An application can query the data in a read-only table snapshot, but cannot insert, update, or delete rows in the snapshot.

Figure 31–5 and the sections that follow explain more about read-only table snapshots and basic replication.

Figure 31–5 Read-only Snapshots, Master Tables, and Snapshot Logs



Read-Only Snapshot Architecture

Oracle supports basic data replication with its table snapshot mechanism. The following sections explain the architecture of simple read-only table snapshots.

> **Note:** Oracle offers other basic replication features such as complex snapshots and ROWID snapshots for unique application requirements. To learn more about these special configurations, see "Other Basic Replication Options" on page 31-10.

A Snapshot's Defining Query The logical data structure of a table snapshot is defined by a query that references data in one or more remote master tables. A snapshot's *defining query* determines what data the snapshot will contain.

A snapshot's defining query should be such that each row in the snapshot corresponds directly to a row or a part of a row in a single master table. Specifically, the defining query of a snapshot should not contain a distinct or aggregate function, a GROUP BY or CONNECT BY clause, join, restricted types of subqueries, or a set operation. The following example shows a simple table snapshot definition.

```
CREATE SNAPSHOT sales.customers AS
    SELECT * FROM sales.customers@hq.acme.com
```

> **Note:** In all cases, the defining query of the snapshot must reference all of the primary key columns in the master table.

A snapshot's defining query can include restricted types of subqueries that reference multiple tables to filter rows from the snapshot's master table. A *subquery snapshot* can be used to create snapshots that "walk up" the many-to-one references from child to parent tables that may involve multiple levels. The following example creates a simple subquery snapshot.

```
CREATE SNAPSHOT sales.orders AS
    SELECT * FROM sales.orders@hq.acme.com o
        WHERE EXISTS
        ( SELECT c_id FROM sales.customers@hq.acme.com c
              WHERE o.c_id = c.c_id AND zip = 19555);
```

Snapshot Refreshes

The data that a snapshot presents does not necessarily match the current data of its master tables. A table snapshot is a transaction-consistent reflection of its master data as that data existed at a specific point in time. To keep a snapshot's data relatively current with the data of its master, Oracle must periodically refresh the snapshot. A *snapshot refresh* is an efficient batch operation that makes that snapshot reflect a more current state of its master.

You must decide how and when it is appropriate to refresh each snapshot to make it a more current representation of its master data. For example, snapshots stemming from master tables that applications often update usually require frequent refreshes. In contrast, snapshots that depend on relatively static master tables usually require infrequent refreshes. In summary, analyze application characteristics and requirements to help determine appropriate snapshot refresh intervals.

To refresh snapshots, Oracle supports different types of refreshes (complete and fast), snapshot refresh groups, and manual and automatic refreshes.

Complete and Fast Refreshes

Oracle can refresh an individual snapshot using either a complete refresh or a fast refresh.

Complete Refreshes To perform a *complete refresh* of a snapshot, the server that manages the snapshot executes the snapshot's defining query. The result set of the query replaces the existing snapshot data to refresh the snapshot. Oracle can perform a complete refresh for any snapshot.

Fast Refreshes To perform a *fast refresh*, the server that manages the snapshot first identifies the changes that took place in the master since the most recent refresh of the snapshot and then applies them to the snapshot. Fast refreshes are more efficient than complete refreshes when there are few changes to the master because participating servers and networks must replicate less data. Fast refreshes are available for snapshots only when the master table has a snapshot log.

Snapshot Logs

When a master table corresponds to one or more snapshots, create a snapshot log for the table so that fast refreshes of the snapshots are an option. A master table's *snapshot log* keeps track of fast refresh data for all corresponding snapshots — only one snapshot log is possible per master table. When a server performs a fast refresh for a snapshot, it uses the data in its master table's snapshot log to refresh the snap-

shot efficiently. Oracle automatically purges specific refresh data from a snapshot log after all snapshots perform refreshes such that the log data is no longer needed.

Snapshot Refresh Groups

To preserve referential integrity and transaction consistency among the table snapshots of several related master tables, Oracle organizes and refreshes each snapshot as part of a *refresh group*. Oracle refreshes all snapshots in a group as a single operation. After refreshing all of the snapshots in a refresh group, the data of all snapshots in the group corresponds to the same transaction consistent point in time.

Automatic Snapshot Refreshes

When creating a snapshot refresh group, administrators usually configure the group so that Oracle automatically refreshes its snapshots. Otherwise, administrators would have to manually refresh the group whenever necessary.

When configuring a refresh group for automatic refreshes, it is necessary to

- specify a refresh interval for the group
- configure the server that manages the snapshots with one or more SNP*n* background processes to wake up periodically and refresh any snapshots that are due for refreshing

Automatic Refresh Intervals When you create a snapshot refresh group, you can specify an automatic refresh interval for the group. When setting a group's refresh interval, understand the following behaviors:

- The dates or date expressions that specify the refresh interval must evaluate to a future point in time.
- The refresh interval must be greater than the length of time necessary to perform a refresh.
- Relative date expressions evaluate to a point in time that is relative to the most recent refresh date. If a network or system failure should interfere with a scheduled group refresh, the evaluation of a relative date expression could change accordingly.
- Explicit date expressions evaluate to a specific point in time, regardless of the most recent refresh date.

Refresh Types By default, Oracle attempts to perform a fast refresh of each snapshot in a refresh group. If, for some reason, Oracle cannot perform a fast refresh for an

individual snapshot (for example, when a master table has no snapshot log), the server performs a complete refresh for the snapshot.

SNP*n* Background Processes Oracle's automatic snapshot refresh facility functions by using job queues to schedule the periodic execution internal system procedures. Job queues require that at least one SNP*n* background process be running. An *SNPn background process* wakes up periodically, checks the job queue, and executes any outstanding jobs.

Manual Snapshot Refreshes

Scheduled, automatic snapshot refreshes may not always be adequate. For example, immediately following a bulk data load into a master table, dependent snapshots will no longer represent the master table's data. Rather than wait for the next scheduled automatic group refreshes, you might want to *manually refresh* dependent snapshot groups to immediately propagate the new rows of the master table to associated snapshots.

Other Basic Replication Options

Oracle supports some additional basic replication features that can be useful in certain situations:

- Complex Snapshots
- ROWID Snapshots

Complex Snapshots

When the defining query of a snapshot contains a distinct or aggregate function, a GROUP BY or CONNECT BY clause, join, restricted types of subqueries, or a set operation, the snapshot is a *complex snapshot*.

The following example is a complex table snapshot definition.

```
CREATE SNAPSHOT scott.emp AS
    SELECT ename, dname
        FROM scott.emp@hq.acme.com a, scott.dept@hq.acme.com b
            WHERE a.deptno = b.deptno
            SORT BY dname
```

The primary disadvantage of a complex snapshot is that Oracle **cannot** perform a fast refresh of the snapshot — Oracle can perform only complete refreshes of a complex snapshot. Consequently, the use of complex snapshots can affect network performance during complete snapshot refreshes.

ROWID Snapshots

Primary key snapshots (discussed implicitly in earlier sections of this chapter) are the default for Oracle. Oracle bases a primary key snapshot on the primary key of its master table. Because of this structure, you can:

- reorganize the master tables of a snapshot without having to complete a full refresh of the snapshot
- create a snapshot with a defining query that includes a restricted type of subquery

For backward compatibility only, Oracle also supports *ROWID snapshots* based on the physical row identifiers (ROWIDs) of rows in the master table. ROWID snapshots should only be used for snapshots of master tables in an Oracle Release 7.3 database, and should not be used when creating new snapshots of master tables in Oracle8 databases.

Note: To support a ROWID snapshot, Oracle creates an additional index on the snapshot's base table with the name *I_SNAP$_snapshotname*.

Advanced Replication Concepts

In advanced replication environments, data replicas anywhere in the system can provide both read and update access to a table's data.

Attention: The information in this section applies only to the advanced replication feature of Oracle8 Enterprise Edition. See *Getting to Know Oracle8 and the Oracle8 Enterprise Edition* for more information about features available with Oracle8 Enterprise Edition.

This section explains the principal concepts of an advanced replication system, including the following topics.

- Uses for Advanced Replication
- Advanced Replication Configurations
- Replication Objects, Groups, Sites, and Catalogs
- Oracle's Advanced Replication Architecture
- Replication Administrators, Propagators, and Receivers

- Replication Conflicts
- Unique Advanced Replication Options

Uses for Advanced Replication

Advanced, symmetric data replication is useful for many types of application systems with special requirements.

Disconnected Environments

Advanced replication is useful for the deployment of transaction processing applications that operate using disconnected components. For example, consider the typical sales force automation system for a life insurance company. Each salesperson must visit customers regularly with a laptop computer and record orders in a personal database while disconnected from the corporate computer network and centralized database system. Upon returning to the office, each salesperson must forward all orders to a centralized, corporate database.

Failover Site

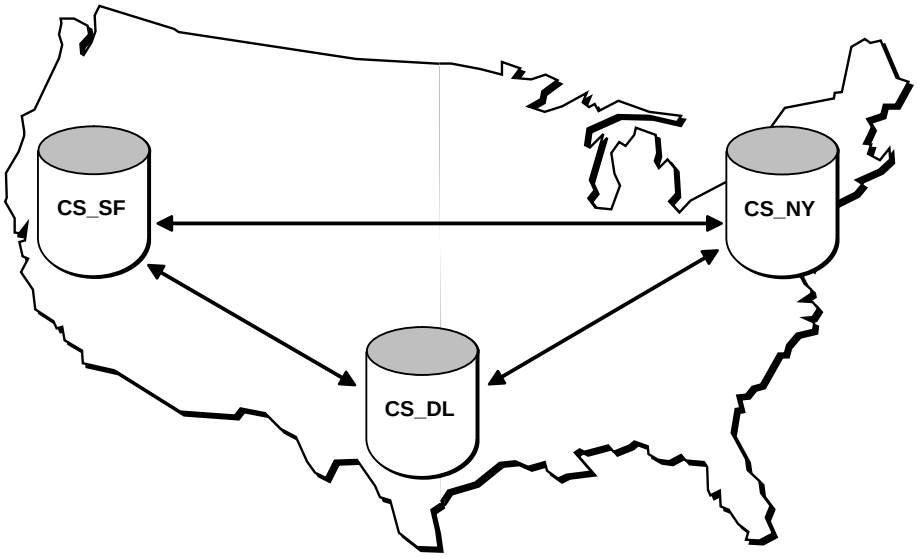
Advanced replication can be useful to protect the availability of a mission critical database. For example, a symmetric replication system can replicate an entire database to establish a failover site should the primary site become unavailable due to a system or network outage. In contrast with Oracle's standby database feature, such a failover site can also serve as a fully functional database to support application access when the primary site is concurrently operational.

Distributing Application Loads

Advanced replication is useful for transaction processing applications that require multiple points of access to database information for the purposes of distributing a heavy application load, ensuring continuous availability, or providing more localized data access.

Applications that have such requirements commonly include customer service oriented applications, as shown in Figure 31–6.

Figure 31–6 Advanced Replication System with Multiple Points of Update Access



Information Transport

Advanced replication can be useful as an information transport mechanism. For example, an advanced replication system can periodically off-load data from an update-intensive operational database to a data warehouse or data mart.

Advanced Replication Configurations

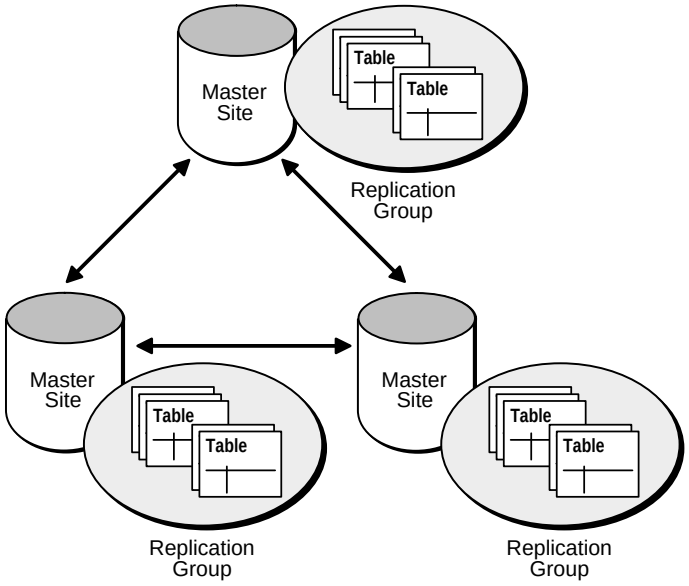
Oracle supports the requirements of advanced replication environments using multimaster replication as well as snapshot sites.

Multimaster Replication

Oracle's *multimaster replication* allows multiple sites, acting as equal peers, to manage groups of replicated database objects. Applications can update any replicated table at any site in a multimaster configuration.

Figure 31–7 illustrates a multimaster symmetric replication system.

Figure 31–7 Multimaster Replication System

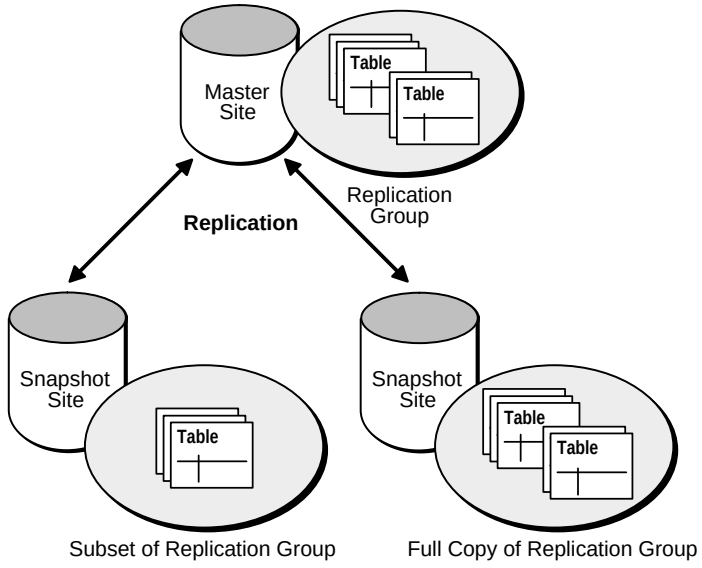


Snapshot Sites and Updatable Snapshots

Master sites in an advanced replication system can consolidate the information that applications update at remote snapshot sites. Oracle's symmetric replication facility allows applications to insert, update, and delete table rows through *updatable snapshots*.

Figure 31–8 illustrates an advanced replication environment with updatable snapshots.

Figure 31–8 Advanced Replication System with Updatable Snapshots



Updatable snapshots have the following properties.

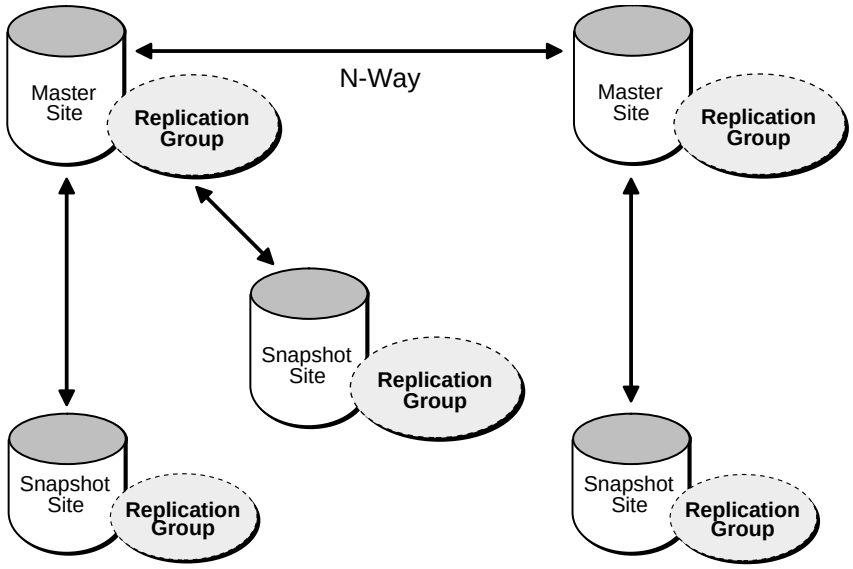
- Updatable snapshots are always simple, fast-refreshable table snapshots.
- Oracle propagates the changes made through an updatable snapshot to the snapshot's remote master table. If necessary, the updates then cascade to all other master sites.
- Oracle refreshes an updatable snapshot as part of a refresh group, identical to read-only snapshots.
- Updatable snapshots have the same underlying objects (base table, indexes, and views) as read-only snapshots. Additionally, Oracle creates a table USLOG$_*snapshotname* to support updatable snapshots.

Hybrid Configurations

Multimaster replication and updatable snapshots can be combined in *hybrid* (mixed) configurations to meet different application requirements. Mixed configurations can have any number of master sites and multiple snapshot sites for each master.

For example, as shown in Figure 31–9, *n*-way replication between two masters can support full-table replication between the databases that support two geographic regions. Snapshots can be defined on the masters to replicate full tables or table subsets to sites within each region.

Figure 31–9 Hybrid Configuration



Some of the key differences between updatable snapshots and replicated masters include the following:

- Replicated masters must contain data for the full table being replicated, whereas snapshots can replicate subsets of master table data.
- Multimaster replication allows you to replicate changes for each transaction as the changes occur. Snapshot refreshes are set oriented, propagating changes from multiple transactions in a more efficient, batch-oriented operation, but at less frequent intervals.
- If any conflicts occur as the result of changes being made to multiple copies of the same data, master sites detect and resolve the conflicts.

Advanced Replication and the Oracle Replication Manager

Advanced replication environments that support an update-anywhere data model can be challenging to configure and manage. To help administer advanced replication environments, Oracle provides a sophisticated management tool, *Oracle Replication Manager*.

> **Additional Information:** *Oracle8 Replication* contains information and examples for using Replication Manager.

Replication Objects, Groups, Sites, and Catalogs

The following sections explain the basic components of an advanced replication system, including replication objects, groups, sites, and catalogs.

Replication Objects

A *replication object* is a database object that exists on multiple servers in a distributed database system. Oracle's advanced replication facility enables you to replicate tables and supporting objects such as views, database triggers, packages, indexes, and synonyms.

Replication Groups

In an advanced replication environment, Oracle manages replication objects using *replication groups*. By organizing related database objects within a replication group, it is easier to administer many objects together. Typically, you create and use a replication group to organize the schema objects necessary to support a particular database application. That is not to say that replication groups and schemas must correspond with one another — the objects in a replication group can originate from several database schemas, and a schema can contain objects that are members of different replication groups. The basic restriction is that a replication object can be a member of only one group.

Replication Sites

A replication group can exist at multiple *replication sites*. Advanced replication environments support two basic types of sites: master sites and snapshot sites.

- A *master site* maintains a complete copy of all objects in a replication group. All master sites in a multi-master, symmetric replication environment communicate directly with one another to propagate data and schema changes in the replication group. A replication group at a master site is more specifically referred to as a *master group*.

- Additionally, every replication group has one and only one *master definition site*. A replication group's master definition site is a master site that serves as the control point for managing the replication group and objects in the group.
- A *snapshot site* supports simple read-only and updatable snapshots of the table data at an associated master site. A snapshot site's table snapshots can contain all or just a subset of the table data within a replication group, but must be simple snapshots that have a one-to-one correspondence to tables at the master site. For example, a snapshot site may contain snapshots for only selected tables in a replication group, and a particular snapshot might be just a selected portion of a certain replicated table. A replication group at a snapshot site is more specifically referred to as a *snapshot group*. A snapshot group can also contain other replication objects.

Replication Catalog

Every master and snapshot site in an advanced replication environment has a *replication catalog*. A site's replication catalog is a distinct set of data dictionary tables and views that maintain administrative information about replication objects and replication groups at the site. Every server that participates in an advanced replication environment can automate the replication of objects in replication groups using the information in its replication catalog.

Replication Management API and Administration Requests

To configure and manage an advanced replication environment, each participating server uses Oracle's replication application programming interface (API). A server's *replication management API* is a set of PL/SQL packages that encapsulate procedures and functions that administrators can use to configure Oracle's advanced replication features. Oracle Replication Manager also uses the procedures and functions of each site's replication management API to perform work.

An *administration request* is a call to a procedure or function in Oracle's replication management API. For example, when you use Replication Manager to create a new master group, Replication Manager completes the task by making a call to the DBMS_REPCAT.CREATE_MASTER_REPGROUP procedure. Some administration requests generate additional replication management API calls to complete the request.

Oracle's Advanced Replication Architecture

Oracle converges the data of typical advanced replication configurations using row-level replication with asynchronous data propagation. The following sections explain how these mechanisms function.

> **Note:** Oracle offers other advanced replication features such as procedural replication and synchronous data propagation for unique application requirements. To learn more about these special configurations, see "Unique Advanced Replication Options" on page 31-26.

Row-Level Replication

Typical transaction processing applications modify small numbers of rows per transaction. Such applications at work in an advanced replication environment will usually depend on Oracle's row-level replication mechanism. With *row-level replication*, applications use standard DML statements to modify the data of local data replicas. When transactions change local data, the server automatically captures information about the modifications and queues corresponding deferred transactions to forward local changes to remote sites.

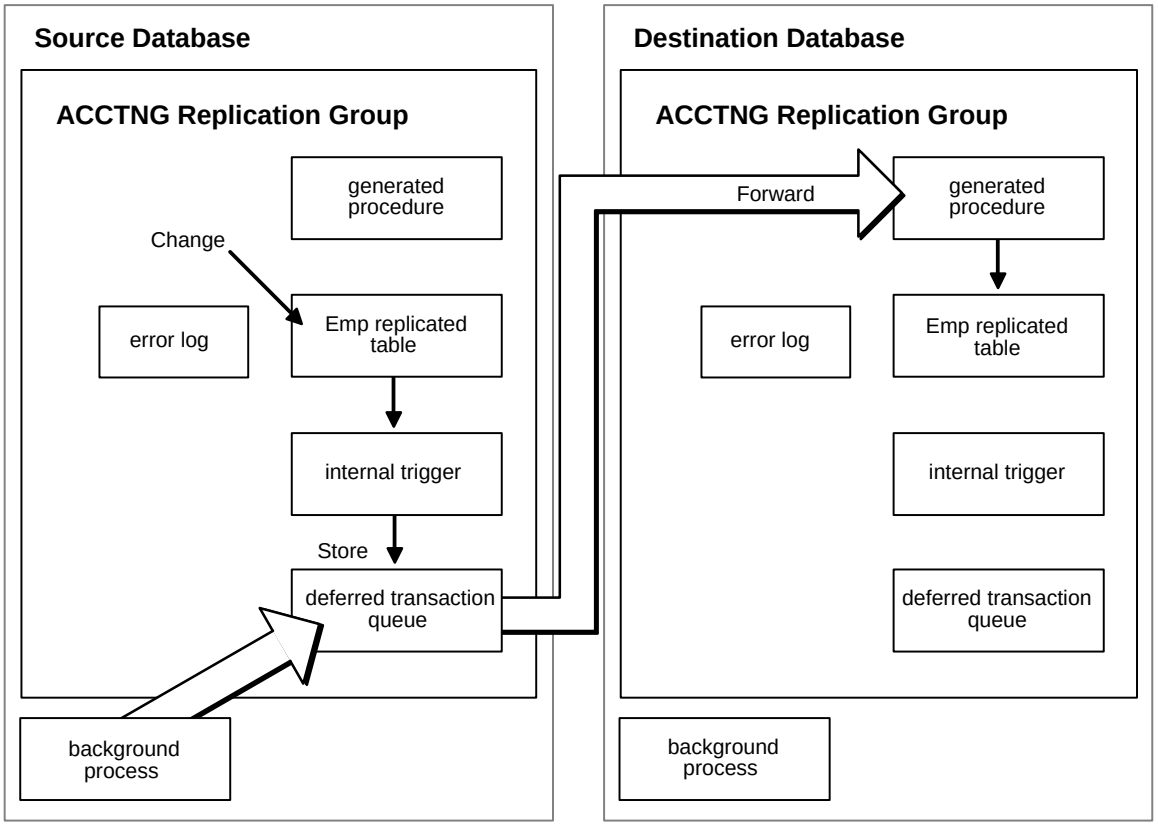
Asynchronous (Store-and-Forward) Data Propagation

Typical advanced replication configurations that rely on row-level replication propagate data level changes using *asynchronous data replication*. Asynchronous data replication occurs when an application updates a local replica of a table, stores replication information in a local queue, and then forwards the replication information to other replication sites at a later time. Consequently, asynchronous data replication is also called *store-and-forward data replication*.

As Figure 31–10 shows, Oracle uses its internal system of triggers, deferred transactions, deferred transaction queues, and job queues to propagate data-level changes asynchronously among master sites in an advanced replication system, as well as from an updatable snapshot to its master table.

- When applications work in an advanced replication environment, Oracle uses internal triggers to capture and store information about updates to replicated data. The internal triggers build *remote procedure calls (RPCs)* that will reproduce the data changes made at the local site to remote replication sites. The internal triggers that support data replication are essentially components within the Oracle server executable; therefore, Oracle can capture and store updates to replicated data very quickly with minimal use of system resources.

Figure 31–10 Asynchronous Data Replication Mechanisms



- Oracle stores RPCs produced by the internal triggers in a site's *deferred transaction queue* for later propagation. Oracle also records information about initiating transactions so that all RPCs that make up a transaction can be propagated and applied remotely as a transaction as well. Oracle's advanced replication facility implements the deferred transaction queue using Oracle's advanced queueing mechanism.
- Oracle manages the propagation process using Oracle's *job queue mechanism* and *deferred transactions*. Each server that participates in an advanced replication system has a local job queue. A server's job queue is a database table that stores information about local jobs such as the PL/SQL call to execute for a job, when to run a job, and so forth. Typical jobs in an advanced replication environment include jobs to push deferred transactions to remote master sites, jobs to

purge applied transactions from the deferred transaction queue, and jobs to refresh snapshot refresh groups.

- Oracle forwards data replication information by executing RPCs as part of deferred transactions. Oracle uses distributed transaction protocols to protect global database integrity automatically and ensure data survivability.

Serial Propagation

With *serial propagation*, Oracle asynchronously propagates replicated transactions, one at a time, in the same order of commit as on the originating site.

Parallel Propagation

With *parallel propagation*, Oracle asynchronously propagates replicated transactions using multiple, parallel transit streams for higher throughput. When necessary, Oracle orders the execution of dependent transactions to ensure global database integrity.

Parallel propagation uses the same execution mechanism that Oracle uses for parallel query, load, recovery, and other parallel operations. Each server process propagates transactions through a single stream. A parallel coordinator process controls these server processes. The coordinator tracks transaction dependencies, allocates work to the server processes, and tracks their progress.

Purging of the Deferred Transaction Queue

After a site pushes a deferred transaction to its destination, the transaction remains in the deferred transaction queue until another job purges the applied transaction from the queue.

Snapshots Propagation Mechanisms

Updatable snapshots in an advanced replication environment can both “push” and “pull” data to and from its master table, respectively.

Master Table Updates Updates to an updatable snapshot are asynchronously pushed to its master table using Oracle’s row-level, asynchronous data propagation mechanisms (RPCs, deferred transactions, and job queues).

Snapshot Refresh Identical to basic replication environments, advanced replication systems use Oracle's snapshot refresh mechanism to pull changes asynchronously from a master table to associated updatable (and read-only) snapshots.

Other Considerations An updatable snapshot's push and pull tasks are independent operations that you can configure associatively or separately.

- A snapshot site can configure a refresh group to automatically push all changes made to the member snapshots to the master site, and then refresh the snapshots.
- A snapshot site can configure updatable snapshots to push changes to the master site and refresh snapshots at different times and intervals.

For example, an advanced replication environment that consolidates information at a master site might configure updatable snapshots to push changes to the master site every hour but refresh updatable snapshots infrequently, if ever.

Replication Administrators, Propagators, and Receivers

An Oracle symmetric replication environment requires several unique database user accounts to function properly, including replication administrators, propagators, and receivers.

- Every site in an Oracle symmetric replication system requires at least one *replication administrator*, a user responsible for configuring and maintaining replicated database objects.
- Each replication site in an Oracle symmetric replication system requires special user accounts to propagate and apply changes to replicated data.

Configuration Options

In most advanced replication configurations, just one account is used for all purposes — as a replication administrator, a replication propagator, and a replication receiver. However, Oracle also supports distinct accounts for unique configurations.

Replication Conflicts

Advanced replication systems that support an update-anywhere model of data replicas must address the possibility of replication conflicts. The following sections explain the different types of replication conflicts, when they can occur, and how Oracle can detect and resolve replication conflicts.

Types of Replication Conflicts

Three types of *conflicts* can occur in an advanced replication environment: uniqueness conflicts, update conflicts, and delete conflicts.

Uniqueness Conflicts A *uniqueness conflict* happens when the replication of a row attempts to violate entity integrity (a PRIMARY KEY or UNIQUE constraint). For example, consider what happens when two transactions that originate from two diffebent sites each insert a row into a respective table replica with the same primary key value — replication of the transactions will cause a uniqueness conflict.

Update Conflicts An *update conflict* happens when the replication of an update to a row conflicts with another update to the same row. Update conflicts can happen when two different transactions, originating from different sites, update the same row at nearly the same time.

Delete Conflicts A *delete conflict* happens when two transactions originate from different sites, with one transaction deleting a row that the other transaction updates or deletes.

Replicated Data Models and Conflicts

When designing applications that will work on top of a database system that uses advanced replication, you must consider the possibility of replication conflicts. In doing so, applications must choose to employ one of several different replicated *data ownership models* that will ensure global database integrity by avoiding or resolving replication conflicts.

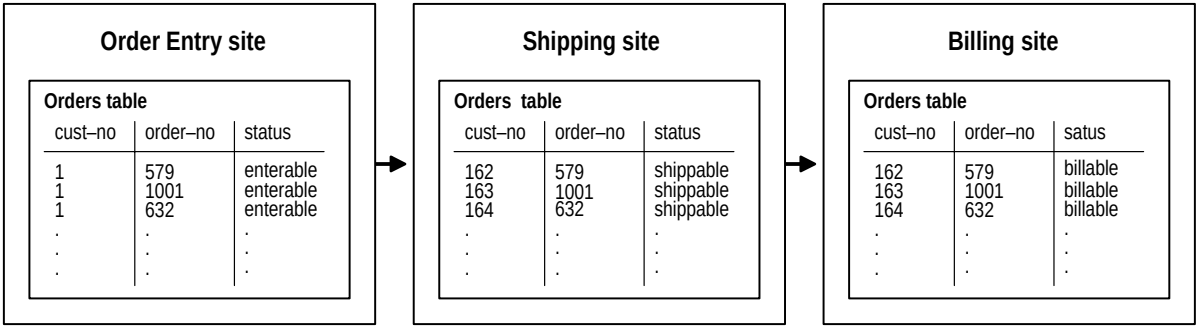
Primary Site, Static Ownership *Primary ownership,* also called *static ownership,* is the replicated data model that basic read-only replication environments support. Primary ownership prevents all replication conflicts, because only a single server permits update access to a set of replicated data.

Rather than control the ownership of data at the table level, applications can employ horizontal and vertical partitioning to establish more granular static ownership of data. For example, applications might have update access to specific columns or rows in a replicated table on a site-by-site basis.

Dynamic Ownership The *dynamic ownership* replicated data model is less restrictive than primary site ownership. With dynamic ownership, the capability to update a data replica moves from site to site, still ensuring that only one site provides update access to specific data at any given point in time. A workflow system clearly illustrates the concept of a dynamic ownership. For example, related departmental applications can read the status code of a product order to determine when they can and cannot update the order.

Figure 31–11 illustrates an application that uses a dynamic ownership model.

Figure 31–11 Dynamic Ownership in an Order Processing System



Shared Ownership Primary site ownership and dynamic ownership replication data models, which promote conflict avoidance, are often too restrictive or impossible to implement for some database applications. Some applications must operate using a *shared ownership* replicated data model in which applications can update the data of any table replica at any time.

When a shared data ownership system replicates changes asynchronously (store-and-forward replication), corresponding applications must be sure to avoid, or detect and resolve replication conflicts if and when they occur. For example:

- Delete conflicts are difficult to resolve in an asynchronous shared ownership model unless the replication system records historical information as transactions delete data. Consequently, applications that operate within an asynchronous, shared ownership data model should avoid delete conflicts by not using DELETE statements to delete rows. Instead, applications can mark rows for deletion and configure the system to periodically purge deleted rows using procedural replication.
- *Coordinated sequence generation* is a technique that applications can use to avoid uniqueness conflicts in a shared data ownership system. For example, typical applications use Oracle sequences to generate numerical primary keys. At each site in a shared data ownership system, create replica sequences so that each sequence generates a mutually exclusive set of sequence numbers.

Conflict Detection

When an application uses a shared ownership data model with asynchronous row-level replication, Oracle automatically detects uniqueness, update, and delete con-

flicts. To detect conflicts during replication, Oracle compares a minimal amount of row data from the originating site with the corresponding row information at the receiving site. When there are differences, Oracle detects the conflict.

To detect replication conflicts accurately, Oracle must be able to uniquely identify and match corresponding rows at different sites during data replication. Typically, Oracle's advanced replication facility uses the primary key of a table to uniquely identify rows in the table. When a table does not have a primary key, you must designate an *alternate key* — a column or set of columns that Oracle can use to identify rows in the table during data replication. In either case, applications should not be allowed to update the identity columns of a table to ensure that Oracle can identify rows and preserve the integrity of replicated data.

Conflict Resolution

When a receiving site in an advanced replication system is using asynchronous row-level replication and it detects a conflict in a transaction, the default behavior is to log the conflict and the entire transaction, and leave the local version of the data intact. In most cases, you should use Oracle's advanced replication facility to automate the resolution of replication conflicts. You can also check each server's DEFERROR data dictionary view for transactions that caused conflicts, and resolve them manually, if necessary.

Column Groups Oracle uses *column groups* to detect and resolve conflicts during asynchronous, row-level symmetric replication. A column group is a logical grouping of one or more columns in a table. Every column in a replicated table is part of a single column group. When configuring replicated tables, you can create column groups and then assign columns and corresponding conflict resolution methods to each group.

Each column group in a replicated table can have a list of one or more conflict resolution methods. Indicating multiple conflict resolution methods for a group allows Oracle to resolve a conflict in different ways should others fail to resolve the conflict. When trying to resolve a conflict for a group, Oracle executes the group's resolution methods in the order that you list for the group.

By default, every replicated table has a *shadow column group*. A table's shadow column group contains all columns that are not within a specific column group. You cannot assign conflict resolution methods to a table's shadow group.

Conflict Resolution Methods When designing column groups you can choose from among many built-in *conflict resolution methods*. For example, to resolve update conflicts, you might choose to have Oracle overwrite the column values at the destina-

tion site with the column values from the originating site. Oracle offers many other conflict resolution methods.

Unique Advanced Replication Options

Some applications have special requirements of an advanced replication system. The following sections explain the Oracle unique advanced replication options, including

- Procedural Replication
- Synchronous (Real-Time) Data Propagation

Procedural Replication

Batch processing applications can change large amounts of data within a single transaction. In such cases, typical row-level replication could saturate a network with a huge quantity of data changes. To avoid such problems, a batch processing application that operates in an advanced replication environment can use Oracle's *procedural replication* to replicate simple stored procedure calls that will converge data replicas. Procedural replication replicates only the call to a stored procedure that an application uses to update a table. Procedural replication does not replicate data modifications.

To use procedural replication, at all sites you must replicate the packages that modify data in the system. After replicating a package, you must generate a *wrapper* for this package at each site. When an application calls a packaged procedure at the local site to modify data, the wrapper ensures that the call is ultimately made to the same packaged procedure at all other sites in the replicated environment. Procedural replication can occur asynchronously or synchronously.

Conflict Detection and Procedural Replication When an advanced replication system replicates data using procedural replication, the procedures that replicate data are responsible for ensuring the integrity of the replicated data. That is, you must design such procedures either to avoid or to detect replication conflicts and resolve them appropriately. Consequently, procedural replication is most typically used when databases are available only for the processing of large batch operations. In such situations, replication conflicts are unlikely because numerous transactions are not contending for the same data.

Additional Information: See *Oracle8 Replication*.

Synchronous (Real-Time) Data Propagation

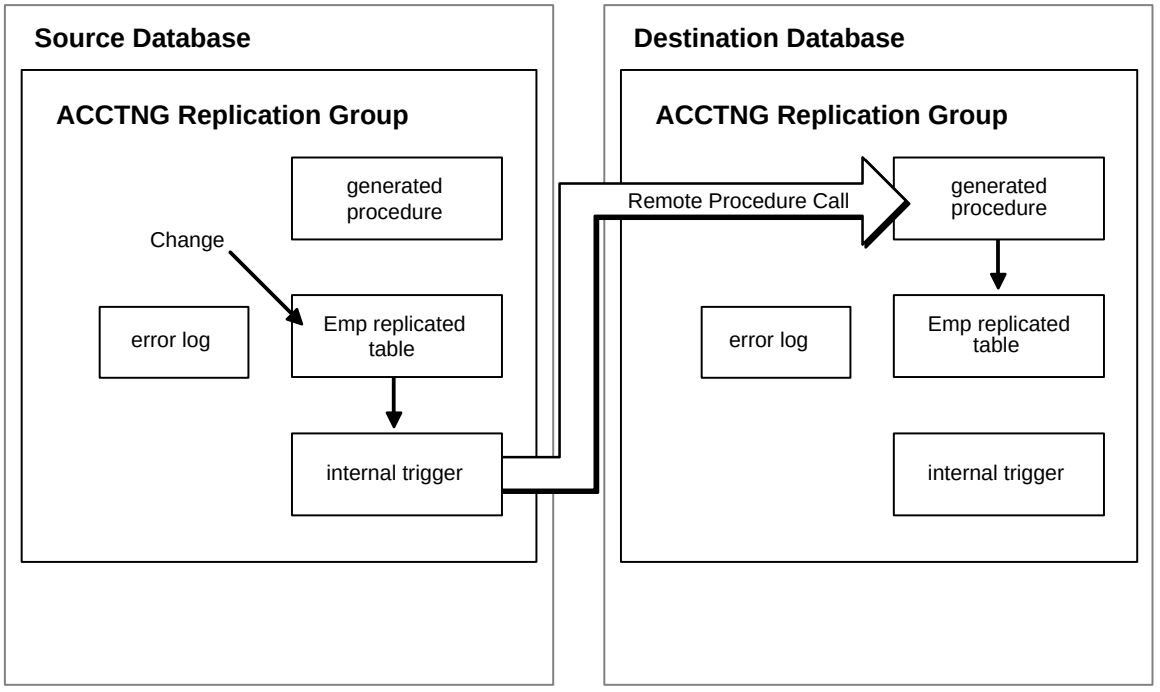
Asynchronous data is the normal configuration for advanced replication environments. However, Oracle also supports synchronous data propagation for applications with special requirements. *Synchronous data propagation* occurs when an application updates a local replica of a table, and within the same transaction also updates all other replicas of the same table. Consequently, synchronous data replication is also called *real-time data replication*. Use synchronous replication only when applications require that replicated sites remain continuously synchronized.

Note: A replication system that uses real-time propagation of replication data is highly dependent on system and network availability because it can function only when all sites in the system are concurrently available.

As Figure 31–12 shows, Oracle uses the same system of internal database triggers to generate RPCs that replicate data-level changes to other replication sites to support synchronous, row-level data replication. However, Oracle does not defer the execution of such RPCs. Instead, data replication RPCs execute within the boundary of the same transaction that modifies the local replica. Consequently, a data-level change must be possible at all sites that manage a replicated table or else a transaction rollback occurs.

You can choose to create a replicated environment in which some sites propagate changes synchronously while others use asynchronous propagation (deferred transactions).

Figure 31–12 Synchronous Data Replication Mechanisms



Replication Conflicts and Synchronous Data Replication When a shared ownership system replicates all changes synchronously (real-time replication), replication conflicts are not possible. With real-time replication, applications use distributed transactions to update all replicas of a table at the same time. As is the case in non-distributed database environments, Oracle automatically locks rows on behalf of each distributed transaction to prevent all types of destructive interference among transactions. While a real-time replication system can prevent replication conflicts, this type of system is highly dependent on system and network availability because it can function only when all sites in the system are available.

> **Additional Information:** See *Oracle8 Replication* for a full description of basic and advanced database replication.

Part IX

Appendix

Part IX contains the following appendix:

- Appendix A, “Operating System-Specific Information”

A

Operating System-Specific Information

This manual occasionally refers to other Oracle manuals that contain detailed information for using Oracle on a specific operating system. These Oracle manuals are often called *installation and configuration guides*, although the exact name may vary on different operating systems. Throughout this manual, references to these manuals are marked with the icon shown in the left margin.

This appendix lists all the references in this manual to operating system-specific Oracle manuals, and lists the operating system (OS) dependent initialization parameters. If you are using Oracle on multiple operating systems, this appendix can help you ensure that your applications are portable across these operating systems.

Operating system-specific topics in this manual are listed alphabetically, with references to the sections that discuss these topics.

- Administrator privileges, prerequisites: “Connecting with Administrator Privileges” on page 5-3
- Auditing: “Auditing to the OS Audit Trail” on page 27-6
- Authenticating users: “Authentication by the Operating System” on page 25-3
- Authenticating database administrators: “Connecting with Administrator Privileges” on page 5-3 and “Database Administrator Authentication” on page 25-6
- Background processes, creating: “Background Processes” on page 7-6
- Background processes, multiple DBWR processes: “Database Writer (DBWn)” on page 7-8
- Background processes, using ARCH: “Archiver Process (ARCH)” on page 7-12
- Client / server communication: “Dedicated Server (Two-Task) Configuration” on page 7-18
- Communication software: “Operating System Communications Software” on page 7-28
- Configuring Oracle: “Variations in Oracle Configuration” on page 7-16
- Datafiles, size of file header: “Size of Datafiles” on page 3-12
- Dedicated server, requesting for administrative operations: “Restricted Operations of the Multithreaded Server” on page 7-24
- Indexes, overhead of index blocks: “Format of Index Blocks” on page 8-19
- Parallel recovery and asynchronous I / O: “Situations That Benefit from Parallel Recovery” on page 28-14
- Role management by the operating system: “The Operating System and Roles” on page 26-16
- Rollback segments, number of transactions per: “Transactions and Rollback Segments” on page 2-18
- Software code areas, shared or unshared: “Software Code Areas” on page 6-16
- Net8, choosing and installing network drivers: “The Program Interface Drivers” on page 7-27
- Net8, drivers included in Net8 software: “How Net8 Works” on page 29-5

Index

A

access control, 26-2
  discretionary, 1-28
  mandatory, 1-34
  password encryption, 25-4
  privileges, 26-2
  roles, 26-10
access paths
  cluster join, 20-49
  composite index, 20-51
  defined, 20-4
  hash cluster key, 20-50
  indexed cluster key, 20-50
  list of, 20-45
  optimization, 20-42
  single row by cluster join, 20-47
  single row by hash cluster key (with unique key), 20-48
  single row by ROWID, 20-47
  single row by unique or primary key, 20-48
ADMIN OPTION
  roles, 26-14
  system privileges, 26-3
  with EXECUTE ANY TYPE, 12-11
administration request, 31-18
administrator privileges
  not audited, 27-5
ADT, *See* object type
Advanced Networking Option, 30-17
  data encryption, 30-17
Advanced Queuing (Oracle AQ), 16-1
  exporting queue tables, 16-8
  message queuing, 16-2
  queue monitor process, 1-19, 7-13, 16-4
    interval statistics, 16-8
    window of execution, 16-5
  queue tables, 16-4
  remote databases, 16-8
advanced replication, 31-3
  asynchronous propagation, 31-19
  deferred transactions and, 31-20
  hybrid configurations, 31-15
  job queues and, 31-20
  multi-master configuration, 31-13
  overview, 31-11
  procedural replication, 31-26
  row-level replication, 31-19
  RPCs and, 31-19
  sequences and, 31-24
  synchronous propagation, 31-27
  updatable snapshots, 31-14
  uses for, 31-12
affinity
  parallel DML, 22-41
  partitions, 22-40
AFTER triggers, 18-9
  defined, 18-9
  when fired, 18-14
agents for queuing, 16-4
ALERT files, 7-14
  redo log, 7-10
aliases
  qualifying column names, 12-2, 12-3
ALL, 20-16
ALL_ views, 4-6
ALL_ROWS hint, 20-42
ALL_UPDATABLE_COLUMNS view, 8-13

ALTER ANY TYPE privilege, 12-10
See also privileges
ALTER command, 14-4
auditing partitions, 9-42
ALTER INDEX command
REBUILD PARTITION, 9-39
ALTER SESSION command, 14-5
dynamic parameters, 5-5
ENABLE PARALLEL DML, 22-32
HASH_JOIN_ENABLED, 20-68
OPTIMIZER_GOAL, 20-41
SET CONSTRAINTS DEFERRED, 24-20
transaction isolation level, 23-7, 23-30
ALTER SYSTEM command, 14-5
dynamic parameters, 5-5
ALTER TABLE command
auditing, 27-7
CACHE clause, 6-4
DEALLOCATE UNUSED, 2-13
disable or enable constraints, 24-21
EXCHANGE PARTITION, 9-10
no-logging mode for SPLIT PARTITION, 21-7
novalidate constraints, 24-21
ALTER USER command
temporary segments and, 2-16
alternate key
detecting conflicts and, 31-25
ALWAYS_ANTI_JOIN parameter, 20-74
ALWAYS_SEMI_JOIN parameter, 20-74
ANALYZE command, 14-4
COMPUTE STATISTICS clause, 20-41
creating histograms, 20-8
ESTIMATE STATISTICS clause, 20-41
partition statistics, 9-12
shared pool and, 6-11
anonymous PL/SQL blocks, 14-15, 17-8
applications, 14-17
calling a stored procedure, 14-18
contrasted with stored procedures, 17-8
dynamic SQL, 14-19
performance, 17-9
ANSI SQL standard
datatypes of, 10-19
Oracle certification, 1-3
ANSI/ISO SQL standard, 1-3
composite foreign keys, 24-15
data concurrency, 23-2
isolation levels, 23-10
anti-joins, 20-74
ANY, 20-15
applications
application vs. database triggers, 18-4
can find constraint violations, 24-6
data dictionary references, 4-4
data warehousing, 8-23, 20-75
database access through, 7-2
decision support systems (DSS), 8-24
parallel SQL, 22-2, 22-26
dependencies of, 19-9
direct-load INSERT, 22-32
discrete transactions, 15-8
enhancing security with, 1-31, 24-5
index-organized tables, 8-30
information retrieval (IR), 8-30
network communication and, 29-5
object dependencies and, 19-11
online analytical processing (OLAP), 8-31
online transaction processing (OLTP)
reverse key indexes, 8-22
parallel DML, 22-31
processes, 7-2, 7-4
program interface and, 7-27
roles and, 26-12
sharing code, 6-17
spatial applications, 8-31
transaction termination and, 15-5
AQ
exporting queue tables, 16-8
message queuing, 16-2
queue monitor process, 1-19, 7-13, 16-4
interval statistics, 16-8
window of execution, 16-5
queue tables, 16-4
remote databases, 16-8
AQ_ADMINISTRATOR role, 16-5
AQ_TM_PROCESS parameter, 16-5
ARCH background process, 7-12
See also archiver process

architecture
client/server, 1-23
MPP, 22-41
of Oracle, 1-13
SMP, 22-41
archived redo log, 1-37
automatic archiving, 28-17
enabling, 28-16
manual archiving, 28-18
ARCHIVELOG mode
archiver process (ARCH) and, 7-12, 28-16
defined, 28-16
overview, 1-38
partial database backups, 1-39, 28-22
whole database backups, 28-21
archiver process (ARCH)
automatic archiving, 28-17
defined, 1-18
described, 7-12
example, 28-16
array processing, 14-13
arrays
size of VARRAYs, 11-10
variable (VARRAYs), 11-10
asynchronous I/O
parallel recovery and, 28-14
asynchronous processing, 16-2
asynchronous replication, 31-19
atomic nulls, 12-6
attributes of object types, 11-2, 11-4
AUDIT command, 14-4
locks, 23-27
audit trail
deleting data in dictionary, 4-5
auditing, 1-33, 27-1
audit options, 27-3
audit records, 27-3
audit trails, 27-3
database, 27-3
operating system, 27-5, 27-6
by access, 27-11
mandated for, 27-11
by session, 27-10
prohibited with, 27-11
data dictionary used for, 4-5
database and OS usernames, 25-4
DDL statements, 27-7
described, 1-33, 27-2
distributed databases and, 27-6
DML statements, 27-7
partitioned tables and indexes, 9-42
privilege use, 27-2, 27-7
range of focus, 27-3, 27-9
schema object, 27-2, 27-8
security and, 27-6
statement, 27-2, 27-7
successful executions, 27-9
transaction independence, 27-5
types of, 27-2
unsuccessful executions, 27-9
user, 27-12
when options take effect, 27-6
authentication
described, 25-3
network, 25-4
operating system, 25-3
Oracle, 25-4
automatic refresh
refresh group, 31-9
refresh interval, 31-9

B

B*-tree indexes, 8-20
bitmap indexes vs., 8-23, 8-24
index-organized tables, 8-28
back-ends, 29-2
background processes, 1-17, 7-6
described, 7-6
diagrammed, 7-6
overview of, 1-17
trace files for, 7-14
See also processes
backups
control files, 28-22
datafiles, 28-22
for read-only tablespaces, 28-23
overview of, 1-34, 28-21
backups *(continued)*

backups *(continued)*
parallel, 28-12
partial, 1-39, 28-22
Recovery Manager, 1-40, 28-10
types of, 1-38
using Export to supplement, 28-23
whole database backup, 1-39, 28-21
base tables, 1-43
data dictionary, 4-2
See also views
basic replication, 31-2, 31-4
uses of, 31-4
BEFORE triggers, 18-8
defined, 18-8
when fired, 18-14
BETWEEN, 20-17
BFILE datatype, 10-10
binary data
BFILEs, 10-10
BLOBs, 10-10
RAW and LONG RAW, 10-11
bind variables
optimization, 20-60
user-defined types, 11-12
bitmap indexes, 8-23
nulls and, 8-8, 8-27
parallel query and DML, 8-24
partitioned tables, 9-12
scans of, 20-45
star transformation, 20-77
BLOBs, 10-10
blocking transactions, 23-10
blocks
anonymous, 14-15, 17-8
database, 2-3
See also data blocks
BOOLEAN datatype, 10-2
branch blocks, 8-21
buffer cache, 6-3, 7-8
extended buffer cache (32-bit), 6-13
multiple buffer pools, 6-5
buffer pools, 6-5
BUFFER_POOL_KEEP parameter, 6-5
BUFFER_POOL_RECYCLE parameter, 6-5
buffers
database buffer cache, 1-15, 6-3, 7-8
incremental checkpoint, 7-9, 28-4
redo log buffer, 1-15, 6-6
business rules
enforcing in application code, 24-5
enforcing using stored procedures, 24-5
enforcing with constraints, 1-54, 24-1
advantages of, 24-5
enforcing with triggers, 1-55

C

CACHE clause, 6-4
caches
buffer cache, 6-3
multiple buffer pools, 6-5
cache hit, 6-4
cache miss, 6-4
data dictionary, 4-4, 6-10
location of, 6-6
database buffer, 1-15
library cache, 6-6
object cache, 11-13, 12-13
object views, 13-4
private SQL area, 6-8
shared SQL area, 6-6, 6-8
writing of buffers, 7-8
calls
Oracle call interface, 7-27
remote procedure, 30-11
cannot serialize access, 23-10
capture avoidance rule, 12-2
Cartesian products, 20-13
CASCADE actions
DELETE statements and, 24-16
catalog, replication, 31-18
century, 10-9
chaining of rows, 2-10, 8-5
CHAR datatype, 10-3
blank-padded comparison semantics, 10-3
character sets
CLOB and NCLOB datatyes, 10-10
column lengths, 10-4
for various languages, 5-4
NCHAR and NVARCHAR2, 10-4

CHECK constraints, 24-16
checking mechanism, 24-19
defined, 24-16
multiple constraints on a column, 24-17
partially null foreign keys, 24-15
partition views, 9-10
subqueries prohibited in, 24-16
checkpoint process (CKPT), 1-18, 7-11
checkpoints
checkpoint process (CKPT), 1-18, 7-11
control files and, 28-20
DBW*n* process, 7-11
incremental, 7-9, 28-4
signal DBW*n* process, 7-8
statistics on, 7-11
CHOOSE hint, 20-42
CKPT background process, 1-18, 7-11
CLEANUP_ROLLBACK_ENTRIES parameter, 22-34
client/server architectures, 29-2
clients, 1-24
diagrammed, 29-3
direct and indirect connections, 30-2
distributed databases and, 30-2
distributed processing in, 29-3
overview of, 1-23, 29-2
program interface, 7-27
CLOB datatype, 10-10
cluster joins, 20-66
cluster keys, 1-45, 8-32
clustered computer systems
Oracle Parallel Server, 5-2
clusters
choosing data to cluster, 8-34
defined, 1-45
dictionary locks and, 23-28
hash, 8-36
allocation of space for, 8-41
collision resolution, 8-39
contrasted with index, 8-36
root blocks, 8-41
scans of, 20-43, 20-48, 20-50
storage of, 8-37
index, 8-35
contrasted with hash, 8-36
scans of, 20-50
indexes and, 8-17
joins and, 8-34, 20-47, 20-49, 20-66
keys, 1-45, 8-32, 8-35
affect indexing of nulls, 8-8
overview of, 8-32
performance considerations of, 8-34
ROWIDs and, 8-7
scans of, 6-4, 20-43, 20-47
hash, 20-48, 20-50
joins, 20-49
setting parameters of, 8-34
storage format of, 8-34
storage parameters of, 8-4
collections, 11-9
nested tables, 11-10
variable arrays (VARRAYs), 11-10
column group, 31-25
shadow, 31-25
columns
column names
qualifying in queries, 12-2, 12-3
column objects, 11-8
indexes, 12-9
default values for, 8-8
defined, 1-42
described, 8-3
integrity constraints, 8-4, 8-8, 24-4, 24-7
maximum in concatenated indexes, 8-18
maximum in view or table, 8-10
nested tables, 8-9
order of, 8-7
prohibiting nulls in, 24-7
pseudocolumns
ROWID, 10-12
ROWNUM, 20-25, 20-34, 20-58
USER, 26-7
COMMENT command, 14-4
COMMIT command, 14-5
ending a transaction, 15-2, 15-4
fast commit, 7-10
implied by DDL, 15-2, 15-4
two-phase commit and, 15-7, 30-12
two-phase commit in parallel DML, 22-34

committing transactions
defined, 15-2
fast commit, 7-10
group commits, 7-10
implementation, 7-10
overview, 1-51
parallel DML, 22-34
communication protocols, 29-5
comparison methods, 11-6
compatibility, 1-4
compilation of object types, 12-14
compiled PL/SQL, 17-15
advantages of, 17-8
procedures, 17-8
pseudocode, 17-16, 18-17
recompiling, 17-17
shared pool, 14-16
triggers, 18-17
compiled triggers, 18-17
complete refresh, 31-8
complex snapshot, 31-10
complex view merging, 20-25
COMPLEX_VIEW_MERGING parameter, 20-26
composite indexes, 8-18
compression of free space in data blocks, 2-9
COMPUTE STATISTICS clause, 20-41
concatenated indexes, 8-18
concurrency
defined, 1-20
described, 23-2
direct-load INSERT, 21-9
enforced with locks, 1-22
limits on
per database, 25-14
per user, 25-12
partition maintenance, 9-33
restrictions on, 1-32, 21-9
transactions and, 23-15
configuration of a database
parameter file, 5-4
process structure, 7-2, 7-16
conflict resolution, 31-25
conflicts, 31-22
column groups and, 31-25
data models and, 31-23
delete, 31-23
detecting, 31-24
procedural replication, 31-26
replication, 31-22
resolving, 31-25
row-level replication, 31-24
uniqueness, 31-23
update, 31-23
CONNECT BY clause
optimizing view queries, 20-25
CONNECT INTERNAL, 5-3
CONNECT role, 26-16
user-defined types, 12-10, 12-11
connectibility, 1-4
connections
defined, 7-4
embedded SQL, 14-6
listener process and, 7-14
restricting, 5-5
sessions contrasted with, 7-4
usernames, 25-2
with administrator privileges, 5-3
consistency of data, 1-51
multiversion consistency model, 1-21
See also read consistency
constants
comparisons and, 20-14
in stored procedures, 14-17
when computed, 20-14
constraints, 1-54
alternatives to, 24-5
applications can find violations, 24-6
CHECK, 24-16
default values and, 24-19
defined, 8-4
disabling temporarily, 24-6
effect on performance, 24-6
enable or disable constraints, 24-21
enforced with indexes, 8-19
PRIMARY KEY, 24-11
UNIQUE, 24-9
FOREIGN KEY, 1-55, 24-12
mechanisms of enforcement, 24-17
NOT NULL, 24-7, 24-10
novalidate constraints, 24-21

constraints *(continued)*
object tables, 12-8
overview, 1-54
parallel create table, 22-22
PRIMARY KEY, 1-55, 24-10
prohibited in views, 8-11
referential
effect of updates, 24-15
self-referencing, 24-14
triggers cannot violate, 18-14
triggers contrasted with, 18-5
types listed, 1-54, 24-1
UNIQUE key, 1-55, 24-8
partially null, 24-10
what happens when violated, 24-5
when evaluated, 8-8
constructor methods, 1-53, 11-6, 12-4
literal invocation of, 12-7
contention
for data
deadlocks, 7-23, 23-16
lock escalation does not occur, 23-16
for rollback segments, 2-19
control files, 1-12, 28-19
backing up, 28-22
changes recorded, 28-20
checkpoints and, 28-20
contents, 28-19
how specified, 5-4
multiplexed, 1-38, 28-20
overview, 1-12, 28-19
physical database structure, 1-5
recovery and, 1-38
used in mounting database, 5-6
converting data
ANSI datatypes, 10-19
program interface, 7-27
SQL/DS and DB2 datatypes, 10-19
coordinated sequence generation, 31-24
cost-based optimization, 20-6
histograms, 20-8
statistics, 20-41
CPU time limit, 25-11
CREATE ANY TYPE privilege, 12-10
See also privileges
CREATE command, 14-4
CREATE FUNCTION command, 17-15
CREATE INDEX command
no-logging mode, 21-7
object types, 12-9
rules of parallelism, 22-21
temporary segments and, 2-16
CREATE PACKAGE BODY command, 17-10, 17-15
CREATE PACKAGE command
examples, 17-10, 18-10
locks, 23-27
package name, 17-15
CREATE PROCEDURE command
example, 17-6
locks, 23-27
procedure name, 17-15
CREATE SYNONYM command
locks, 23-27
CREATE TABLE AS SELECT
direct-load INSERT vs., 21-2
no-logging mode, 21-7
rules of parallelism, 22-22
space fragmentation, 22-27
CREATE TABLE command
auditing, 27-7, 27-9
CACHE clause, 6-4
enable or disable constraints, 24-21
examples
column objects, 11-5, 12-2
nested tables, 11-11
object tables, 11-7, 11-11, 12-2, 12-8
locks, 23-27
parallelism, 22-26
CREATE TRIGGER command
compiled and stored, 18-17
examples, 18-10, 18-12, 18-16
object tables, 12-9
locks, 23-27
CREATE TYPE command
incomplete types, 12-13
nested tables, 11-4, 11-11, 12-7
object types, 11-4, 12-2, 12-6, 12-7
object views, 13-3
VARRAYs, 11-10

CREATE TYPE privilege, 12-10
See also privileges
CREATE USER command
temporary segments and, 2-16
CREATE VIEW command
examples, 18-12
object views, 13-3
locks, 23-27
cross joins, 20-13
cursors
creating, 14-10
defined, 14-6
embedded SQL, 14-6
maximum number of, 14-7
object dependencies and, 19-8
opening, 6-9, 14-7
overview of, 1-15
private SQL areas and, 6-9, 14-6
recursive, 14-7
recursive SQL and, 14-7
stored procedures and, 14-17

D

dangling REFs, 11-8, 11-9
data
access to, 1-48
control of, 25-2
message queues, 16-5
security domains, 25-2
concurrent access to, 23-2
consistency of
defined, 1-51
examples of lock behavior, 23-30
locks, 23-3
manual locking, 23-29
read consistency, 1-21
repeatable reads, 23-6
transaction level, 23-6
underlying principles, 23-14
distributed manipulation of, 1-25
how stored in tables, 8-4
integrity of, 1-20, 8-4, 24-2
CHECK constraints, 24-16
enforcing, 24-4, 24-5
overview, 1-54
parallel DML restrictions, 22-38
referential, 24-3
two-phase commit, 1-25
types, 24-2
locks on, 23-19
replicating, 1-26, 31-2
data blocks, 1-10, 2-2
allocating for extents, 2-11
cached in memory, 7-8
clustered, 8-34
coalescing free, 2-12
controlling free space in, 2-5
format, 2-3
free lists and, 2-9
hash keys and, 8-41
how rows stored in, 8-5
overview, 2-2
read-only transactions and, 23-30
row directory, 8-6
shared in clusters, 8-32
shown in ROWIDs, 10-13, 10-14
space available for inserted rows, 2-9
stored in the buffer cache, 6-3
writing to disk, 7-8
data conversion
ANSI datatypes, 10-19
program interface, 7-27
SQL/DS and DB2 datatypes, 10-19
Data Definition Language (DDL)
auditing, 27-7
commit implied by, 15-4
defined, 1-49
described, 14-4
locks, 23-26
parallel DDL, 22-3
parsing with DBMS_SQL, 14-19
processing statements, 14-14
roles and privileges, 26-14
data dictionary
access to, 4-2
adding objects to, 4-4
ALL prefixed views, 4-6
audit trail (SYS.AUD$), 4-5
backups, 28-23

data dictionary *(continued)*
cache, 6-10
location of, 6-6
content of, 4-2, 6-10
procedures, 17-16
DBA prefixed views, 4-6
defined, 1-47, 4-2
dependencies tracked by, 19-3
DUAL table, 4-7
dynamic performance tables, 4-7
locks, 23-26
owner of, 4-3
prefixes to views of, 4-5
public synonyms for, 4-4
row cache and, 6-10
statistics in, 20-41
partition statistics, 9-12
structure of, 4-2
updates of, 4-5
USER prefixed views, 4-6
uses of, 4-3
table and column definitions, 14-11
validity of procedures, 17-16
views used in optimization, 20-7
data locks
conversion, 23-16
duration of, 23-15
escalation, 23-16
Data Manipulation Language (DML)
auditing, 27-7
defined, 1-49
described, 14-3
distributed transactions, 30-10
locks acquired by, 23-24
parallel DML, 22-3, 22-29
partition locks, 9-30
privileges controlling, 26-5
processing statements, 14-10
serializable isolation for subqueries, 23-13
transaction model for parallel DML, 22-33
triggers and, 18-3, 18-16
data models, 1-40
data object number
extended ROWID, 10-13
data ownership models, 31-23
dynamic ownership, 31-23
primary ownership, 31-23
shared ownership, 31-24
static ownership, 31-23
data segments, 1-11, 2-15, 8-4
data warehousing, 20-75
bitmap indexes, 8-23
refreshing table data, 22-31
star queries, 20-75
database administrators (DBAs)
authentication, 25-6
data dictionary views, 4-6
DBA role, 12-10, 26-16
password files, 25-7
responsible for backup and recovery, 28-2
database buffers
after committing transactions, 15-6
buffer cache, 6-3, 7-8
clean, 7-8
committing transactions, 7-10
defined, 1-15, 6-3
dirty, 6-3, 7-8
free, 6-3
multiple buffer pools, 6-5
pinned, 6-3
size of cache, 6-5
writing of, 7-8
database links, 1-47
defined, 1-47
overview of, 30-6
database management system (DBMS), 1-2
object-relational DBMS, 11-2
Oracle server, 1-4
principles, 1-40
database triggers, 1-55, 18-1
See also triggers
database writer process (DBW*n*), 7-8
checkpoints signal, 7-8
defined, 7-8
least recently used algorithm (LRU), 7-8
media failure, 28-6
multiple DBW*n* processes, 7-8
multiple I/O processes, 7-9
overview of, 1-17

database writer process (DBW*n*) *(continued)*
trace file, 28-6
when active, 7-8
write-ahead, 7-10
writing to disk at checkpoints, 7-11
databases
access control
overview, 1-48
password encryption, 25-4
security domains, 25-2
backing up, 1-39, 28-21
closing, 5-8
aborting the instance, 5-8
configuring, 5-4
contain schemas, 25-2
defined, 1-8
dismounting, 5-8
distributed, 1-24, 30-1
changing global database name, 6-11
nodes of, 1-24, 30-2
overview of, 1-23, 1-24, 30-1
site autonomy of, 30-15
statement optimization on, 20-39
table replication, 1-26
two-phase commit, 1-25
global database names, 30-4
limitations on usage, 25-10
logical structure of, 1-6
logical structures (objects) in, 1-8
modes of archiving, 28-16
mounting, 5-6
name stored in control file, 28-19
open and closed, 5-2
opening, 5-7
acquiring rollback segments, 2-23
physical structure, 1-5, 1-11, 2-2
revealing with ROWIDs, 10-14
recovery of, 1-34, 28-2
scalability, 22-2, 22-31, 29-4
shutting down, 5-8
size of
how determined, 3-6
standby, 28-24
starting up, 5-2
forced, 5-9
datafiles
backing up, 28-22
contents of, 3-12
dictionary in datafile 1, 28-23
in online or offline tablespaces, 3-12
named in control files, 28-19
overview of, 1-9, 1-11, 3-11
parallel recovery, 28-13
physical database structure, 1-5
read-only, 3-9
recovery, 28-5
read-only tablespaces and, 3-10
relationship to tablespaces, 3-2
shown in ROWIDs, 10-13, 10-14
taking offline, 3-12
unrecoverable, 28-13
datatypes, 10-2, 10-17
ANSI, 10-19
array types, 11-10
BOOLEAN, 10-2
CHAR, 10-3
character, 10-2, 10-10
collections, 11-9
conversions of
by program interface, 7-27
non-Oracle types, 10-19
Oracle to another Oracle type, 10-20
DATE, 10-7
DB2, 10-19
how they relate to tables, 8-3
in PL/SQL, 10-2
list of available, 10-2
LOB datatypes, 10-9
BFILE, 10-10
BLOB, 10-10
CLOB and NCLOB, 10-10
default logging mode, 21-7
LONG, 10-5
storage of, 8-7
MLSLABEL, 10-16
multimedia, 11-3
NCHAR and NVARCHAR2, 10-4
nested tables, 8-9, 11-10
NUMBER, 10-5
object types, 1-41, 11-4

datatypes *(continued)*
of columns, 1-42
RAW and LONG RAW, 10-11
REF, 11-8
ROWID, 10-12
SQL/DS, 10-19
summary, 10-17
user-defined, 11-1, 11-3
VARCHAR, 10-3
VARCHAR2, 10-3
DATE datatype, 10-7
arithmetic with, 10-9
changing default format of, 10-7
Julian dates, 10-8
partitioning, 9-12, 9-16
DB_BLOCK_BUFFERS parameter
buffer cache and, 6-5
system global area size and, 6-12
DB_BLOCK_LRU_LATCHES parameter, 7-8
DB_BLOCK_MAX_DIRTY_TARGET parameter, 7-9, 28-5
DB_BLOCK_SIZE parameter
buffer cache and, 6-5
system global area size and, 6-12
DB_FILE_MULTIBLOCK_READ_COUNT parameter, 20-59
cost-based optimization, 20-70
DB_FILES parameter, 6-15
DB_NAME parameter, 28-20
DB_WRITER_PROCESSES parameter, 1-17, 7-8
DBA role, 26-16
user-defined types, 12-10
DBA_ views, 4-6
DBA_QUEUE_SCHEDULES view, 16-8
DBA_SYNONYMS.SQL script
using, 4-6
DBA_UPDATABLE_COLUMNS view, 8-13
DBMS, 1-2
general requirements, 1-48
object-relational DBMS, 11-2
DBMS_AQ package, 16-4
DBMS_AQADM package, 16-4, 16-5
DBMS_JOB package, 7-13
DBMS_LOCK package, 23-40
DBMS_SQL package, 14-19
parsing DDL statements, 14-19
DBW*n* background process, 7-8
See also database writer process
DBWR_IO_SLAVES parameter, 7-9
DDL, 1-49, 14-4
See also Data Definition Language
deadlocks
artificial, 7-23
avoiding, 23-18
defined, 23-16
detection of, 23-17
distributed transactions and, 23-17
deallocating extents, 2-13
decision support systems (DSS), 9-4
bitmap indexes, 8-24
disk striping, 22-41
parallel DML, 22-31
parallel SQL, 22-2, 22-26, 22-31
partitions, 9-4
performance, 9-7, 22-31
scoring tables, 22-32
dedicated servers, 7-18
defined, 1-17
examples of use, 7-25
multithreaded servers vs., 7-20
default values, 8-8
constraints effect on, 8-8, 24-19
user-defined types, 12-7
deferred constraints
deferrable or nondeferrable, 24-20
initially deferred or immediate, 24-20
deferred transactions, 31-20
DefError view
conflicts and, 31-25
define phase of query processing, 14-12
defining query of a snapshot, 31-7
degree of parallelism, 22-17, 22-19
between query operations, 22-11
parallel SQL, 22-7, 22-13
delete cascade constraint, 24-16
DELETE command, 14-4
foreign key references and, 24-15
freeing space in data blocks, 2-9

DELETE command *(continued)*
parallel DELETE, 22-18
triggers and, 18-2, 18-6
INSTEAD OF triggers, 18-11
delete conflict, 31-23
delete no action constraint, 24-15
DELETE privilege for object tables, 12-12, 12-13
dependencies
between schema objects, 19-2
local, 19-9
non-existent referenced objects and, 19-7
object type definitions, 12-13, 12-15
on non-existence of other objects, 19-7
Oracle Forms triggers and, 19-11
privileges and, 19-6
remote objects and, 19-8
shared pool and, 19-8
dereferencing, 11-9
implicit, 11-9
describe phase of query processing, 14-12
dictionary
See data dictionary
dictionary cache locks, 23-29
different row-writers block writers, 23-10
Digital POLYCENTER Manager on NetView, 30-19
direct-load INSERT, 21-2
logging mode, 21-5
parallel INSERT, 21-3
parallel load vs. parallel INSERT, 21-2
restrictions, 21-9, 22-37
serial INSERT, 21-3
space management, 21-8
dirty buffer, 6-3
incremental checkpoint, 7-9, 28-4
dirty read, 23-2, 23-10
dirty write, 23-10
disable constraints, 24-21
disaster recovery, 28-24
discrete transaction management, 15-8
discretionary access control, 1-28, 25-2
disk affinity
parallel DML, 22-41
partitions, 22-40
disk failures, 1-36, 28-5
disk space
controlling allocation for tables, 8-4
datafiles used to allocate, 3-11
disk striping
affinity, 22-40
partitions, 9-8
dispatcher processes (D*nnn*)
defined, 1-18
described, 7-13
limiting SGA space per session, 25-12
listener process and, 7-14
network protocols and, 7-14
prevent startup and shutdown, 7-24
response queue and, 7-21
user processes connect via Net8, 7-14, 7-20
DISTINCT operator
optimizing views, 20-25
distributed databases, 30-1
auditing and, 27-6
client/server architectures and, 29-2
database links, 30-6
deadlocks and, 23-17
dependent schema objects and, 19-8
diagrammed, 30-3
different Oracle versions, 30-7
distributed queries, 30-10
distributed updates, 30-10
global schema object names, 30-6
heterogeneous, 30-8
job queue processes (SNP*n*), 1-19, 7-13
management tools, 30-17
message propagation, 16-8
nodes of, 30-2
overview of, 1-24, 30-2
recoverer process (RECO) and, 7-12
remote dependencies, 19-9
remote queries and updates, 30-10
server can also be client in, 29-2
site autonomy of, 30-15
statement optimization on, 20-39
table replication, 1-26, 31-2, 31-3
for read and update, 31-11
for read only, 31-2, 31-4
transparency of, 30-13
two-phase commit, 1-25, 30-12

distributed processing environment
client/server architecture in, 1-23, 29-3
data manipulation statements, 14-10
described, 1-23, 29-2
distributed databases vs., 30-7
distributed transactions
defined, 30-11
optimizing, 20-39
parallel DML restrictions, 22-40
routing statements to nodes, 14-11
two-phase commit and, 1-25, 15-7
DISTRIBUTED_TRANSACTIONS parameter, 7-12
DML, 1-49, 14-3
See also Data Manipulation Language
D*nnn* background processes, 7-13
See also dispatcher processes
drivers, 7-27
DROP ANY TYPE privilege, 12-10
See also privileges
DROP command, 14-4
DROP TABLE command
auditing, 27-7
DROP TYPE command
dependencies and, 12-15
FORCE option, 12-15
DSS database
disk striping, 22-41
parallel DML, 22-31
partitioning indexes, 9-29
partitions, 9-5
performance, 9-7
scoring tables, 22-32
DUAL table, 4-7
dump files
Export and Import, 12-15
dynamic ownership, 31-23
dynamic partitioning, 22-6
dynamic performance tables (V$ tables), 4-7
dynamic SQL
DBMS_SQL package, 14-19

E

embedded SQL statements, 1-49, 14-5
enable constraints, 24-21
encryption, 30-17
Enterprise Manager
advanced queuing, 16-9
ALERT file, 7-15
checkpoint statistics, 7-11
distributed databases, 30-18
executing a package, 17-6
executing a procedure, 17-4
granting roles, 26-13
granting system privileges, 26-3
lock and latch monitors, 23-28
parallel recovery, 28-13
PL/SQL, 14-17, 14-18
schema object privileges, 26-4
showing size of SGA, 6-12
shutdown, 5-8, 5-9
SQL statements, 14-2
startup, 5-5
statistics monitor, 25-13
equijoins
cluster joins, 20-66
defined, 20-13
hash joins, 20-68
sort-merge, 20-65
equipartitioning, 9-18
errors
in embedded SQL, 14-6
tracked in trace files, 7-14
ESTIMATE STATISTICS clause, 20-41
exceptions
during trigger execution, 18-15
raising, 14-18
stored procedures and, 14-18
EXCHANGE PARTITION, 9-10
exclusive locks
row locks (TX), 23-19
RX locks, 23-22
table locks (TM), 23-20
exclusive mode, 2-24, 5-6
EXECUTE ANY TYPE privilege, 12-10, 12-11
See also privileges

EXECUTE privilege
user-defined types, 12-11, 12-12, 12-13
verifying user access, 17-16
See also privileges
EXECUTE user-defined type, 12-10
execution plan
accessing views, 20-28, 20-31, 20-32
complex statements, 20-23
compound queries, 20-36, 20-37, 20-38
joining views, 20-34
joins, 20-63, 20-69
OR operators, 20-20
star transformation, 20-78
execution plans
examples, 20-23
execution sequence of, 20-5
EXPLAIN PLAN, 14-4
location of, 6-8
overview of, 20-2
parsing SQL, 14-11
partitions and partition views, 9-10, 9-12
viewing, 20-4
EXP_FULL_DATABASE role, 26-16
EXPLAIN PLAN command, 14-4
access paths, 20-47, 20-48, 20-49, 20-50, 20-51, 20-52, 20-53, 20-54, 20-55, 20-56, 20-57, 20-58
star query, 20-77
star transformation, 20-78
explicit locking, 23-29
Export utility
partition maintenance operations, 9-31
use in backups, 28-23
user-defined types, 12-15
extended ROWID format, 10-12
extents
allocating data blocks for, 2-11
allocation to rollback segments
after segment creation, 2-21
at segment creation, 2-19
allocation, how performed, 2-11
as collections of data blocks, 2-10
deallocation
from rollback segments, 2-22
when performed, 2-13
defined, 2-3
dropping rollback segments and, 2-22
in rollback segments
changing current, 2-20
incremental, 2-11
overview of, 2-10
parallel DDL, 22-27
external procedures, 14-19, 17-9

F

failover database, 31-12
failures, 28-2
archiving redo log files, 28-18
database buffers and, 28-8
described, 1-35, 28-2
instance, 1-36, 28-4
recovery from, 28-4
internal errors
tracked in trace files, 7-14
media, 1-36, 28-5
network, 28-3
safeguards provided, 28-7
statement and process, 1-35, 7-12, 28-2
survivability, 28-24
user error, 1-35, 28-2
See also recovery
fast commit, 7-10
fast full index scans, 20-44
fast refresh, 31-8
fast transaction rollback, 28-10
fast warmstart, 28-4
FAST_FULL_SCAN_ENABLED parameter, 20-44
fetching rows in a query, 14-13
embedded SQL, 14-6
file management locks, 23-29
files
ALERT and trace files, 7-10, 7-14
Export and Import dump file, 12-15
initialization parameter, 5-4, 5-5
operating system, 1-5
Oracle database, 1-9, 1-11, 28-7
password, 25-7
administrator privileges, 5-3
See also control files, datafiles, redo log files
FIPS standard, 1-3, 14-6

FIRST_ROWS hint, 20-42
flagging of nonstandard features, 1-3, 14-6
FORCE option
  object type dependencies, 12-15
FOREIGN KEY constraints
  changes in parent key values, 24-15
  constraint checking, 24-19
  deleting parent table rows and, 24-16
  maximum number of columns in, 24-12
  nulls and, 24-14
  updating parent key tables, 24-15
foreign keys, 1-54
  defined, 1-55
  partially null, 24-15
  privilege to use parent key, 26-5
fragmentation
  parallel DDL, 22-28
free lists, 2-9
free space (section of data blocks), 2-5
front-ends, 29-2
full index scans, 20-44
full table scans, 20-43, 20-57
  LRU algorithm and, 6-4
  multiblock reads, 20-59
  parallel query, 22-5, 22-6
  rule-based optimizer, 20-62
  selectivity and, 20-59
functions
  hash functions, 8-40
  PL/SQL, 17-2, 17-6
    contrasted with procedures, 1-52, 17-2
    parallel DML restrictions, 22-39
    privileges for, 26-7
    roles disabled in, 26-14
    See also procedures
  SQL, 14-2
    COUNT, 8-27
    default column values, 8-8
    in CHECK constraints, 24-16
    in views, 8-12
    NVL, 8-7
    optimizing view queries, 20-25, 20-32
fuzzy reads, 23-3

G

gateways, 30-8
global database names
  shared pool and, 6-11
global indexes, 9-25, 9-27
  managing partitions, 9-39
global schema object names, 1-47, 30-6
GRANT ANY PRIVILEGE system privilege, 26-3
GRANT command, 14-4
  locks, 23-27
GRANT option for EXECUTE privilege, 12-11
granting
  execute user-defined type, 12-11
  privileges and roles, 26-3
GROUP BY clause
  optimizing views, 20-25
group commits, 7-10
groups, instance, 22-16

H

handles for SQL statements, 1-15, 6-9
hash clusters, 1-47, 8-36
  overview of, 1-47
  scans of, 20-43, 20-48, 20-50
hash join, 20-68
  HASH_AREA_SIZE parameter, 20-69
  HASH_MULTIBLOCK_IO_COUNT parameter, 20-69
HASH_AJ hint, 20-74
HASH_AREA_SIZE parameter, 20-69
HASH_JOIN_ENABLED parameter, 20-68
HASH_MULTIBLOCK_IO_COUNT parameter, 20-69
HASH_SJ hint, 20-74
HASHKEYS parameter, 8-39
headers
  of data blocks, 2-4
  of row pieces, 8-5
heterogeneous distributed databases, 30-8
Heterogeneous Services, 30-10
HI_SHARED_MEMORY_ADDRESS parameter, 6-13

HIGH_VALUE column
of USER_TAB_COLUMNS view, 20-60
hints
INDEX, 20-76
INDEX_FFS, 20-44
MERGE, 20-26
MERGE_AJ and HASH_AJ, 20-74
MERGE_SJ and HASH_SJ, 20-74
ORDERED, 20-70, 20-76
overriding OPTIMIZER_MODE and OPTIMIZER_GOAL, 20-42
PARALLEL, 22-13
PARALLEL_INDEX, 22-13
PUSH_JOIN_PRED, 20-73
STAR, 20-76
USE_HASH, 20-68
histograms, 20-8
historical database
maintenance operations, 9-32
partitions, 9-5
HP OpenView, 30-19
hybrid configurations
advanced replication, 31-15

I

IBM NetView/6000, 30-19
identity column
detecting conflicts and, 31-25
immediate constraints, 24-19
IMP_FULL_DATABASE role, 26-16
implicit dereferencing, 11-9
Import utility
partition maintenance operations, 9-31
use in recovery, 28-23
user-defined types, 12-15
IN operator, 20-15
merging views, 20-26
IN subquery, 20-25
incomplete object types, 12-14
incremental checkpoint, 7-9, 28-4
index segments, 1-10, 2-15
INDEX_FFS hint, 20-44
indexes, 1-45, 8-17
auditing partitions, 9-42
B*-tree structure of, 8-20
bitmap indexes, 8-23, 8-28
nulls and, 8-8
parallel query and DML, 8-24
branch blocks, 8-21
building
using an existing index, 8-17
cluster, 8-35
contrasted with table, 8-36
dropping, 8-36
scans of, 20-50
composite, 8-18
scans of, 20-51
concatenated, 8-18
described, 1-45, 8-17
enforcing integrity constraints, 24-9, 24-11
fast full scans of, 20-44
global indexes, 9-25, 9-39
index unusable (IU), 9-39
index-organized tables, 8-28
internal structure of, 8-20
keys and, 8-19
primary key constraints, 24-11
unique key constraints, 24-9
leaf blocks, 8-21
local indexes, 9-23, 9-38
location of, 8-19
LONG RAW datatypes prohibit, 10-11
managing partitions, 9-38
no-logging mode, 21-7
non-unique, 8-17
nulls and, 8-8, 8-27
on attribute of object column, 12-9
on object identifiers, 12-5
on REFs, 12-9
optimization and, 20-19
overview of, 1-45, 8-17
parallel DDL storage, 22-27
parallel index scans, 22-5
partition pruning, 9-4
partitioned tables, 8-28
partitioning guidelines, 9-28
partitions, 9-2, 9-22

indexes *(continued)*
performance and, 8-17
privileges for partitions, 9-41
range scans, 20-44
rebuild partition, 9-39
rebuilt after direct-load INSERT, 21-8
reverse key indexes, 8-22
ROWIDs and, 8-21
scans of, 20-43
bounded range, 20-53
cluster key, 20-50
composite, 20-51
MAX or MIN, 20-55
ORDER BY, 20-56
restrictions, 20-57
single-column, 20-51
unbounded range, 20-54
statement conversion and, 20-19
storage format of, 8-19
unique, 8-17
unique scans, 20-44
user-defined types, 12-9
when used with views, 8-12
index-organized tables, 8-28
applications, 8-30
benefits, 8-29
queue tables, 16-9
row overflow area, 8-29
in-doubt transactions, 2-21, 5-7
information consolidation
advanced replication and, 31-14
Information Retrieval (IR) applications
index-organized tables, 8-30
initialization parameters
ALWAYS_ANTI_JOIN, 20-74
ALWAYS_SEMI_JOIN, 20-74
AQ_TM_PROCESS, 16-5
BUFFER_POOL_KEEP, 6-5
BUFFER_POOL_RECYCLE, 6-5
CLEANUP_ROLLBACK_ENTRIES, 22-34
COMPLEX_VIEW_MERGING, 20-26
DB_BLOCK_BUFFERS, 6-5, 6-12
DB_BLOCK_LRU_LATCHES, 7-8
DB_BLOCK_MAX_DIRTY_TARGET, 7-9, 28-5
DB_BLOCK_SIZE, 6-5, 6-12
DB_FILE_MULTIBLOCK_READ_COUNT, 20-59, 20-70
DB_FILES, 6-15
DB_NAME, 28-20
DB_WRITER_PROCESSES, 1-17, 7-8
DBWR_IO_SLAVES, 7-9
DISTRIBUTED_TRANSACTIONS, 7-12
FAST_FULL_SCAN_ENABLED, 20-44
HASH_AREA_SIZE, 20-69
HASH_JOIN_ENABLED, 20-68
HASH_MULTIBLOCK_IO_COUNT, 20-69
HI_SHARED_MEMORY_ADDRESS, 6-13
JOB_QUEUE_PROCESSES, 16-8
LGWR_IO_SLAVES, 7-11
LICENSE_MAX_SESSIONS, 25-14
LICENSE_SESSIONS_WARNING, 25-15
LOCK_SGA, 6-12, 6-16
LOCK_SGA_AREAS, 6-12, 6-16
LOG_ARCHIVE_START, 28-17
LOG_BUFFER, 6-6, 6-12
LOG_CHECKPOINT_INTERVAL, 7-8
LOG_CHECKPOINT_TIMEOUT, 7-8
LOG_FILES, 6-15
MTS_MAX_SERVERS, 7-23, 7-24
MTS_SERVERS, 7-23
NLS_LANGUAGE, 9-15
NLS_NUMERIC_CHARACTERS, 10-6
NLS_SORT, 9-15
OPEN_CURSORS, 6-9, 14-7
OPEN_LINKS, 6-15
OPTIMIZER_MODE, 20-40
PARALLEL_DEFAULT_MAX_SCANS (obsolete), 22-15
PARALLEL_DEFAULT_SCANSIZE (obsolete), 22-15
PARALLEL_MAX_SERVERS, 22-8
PARALLEL_MIN_PERCENT, 22-15
PARALLEL_MIN_SERVERS, 22-7, 22-8
PARALLEL_SERVERS_IDLE_TIME, 22-8
PUSH_JOIN_PREDICATE, 20-73
REMOTE_DEPENDENCIES_MODE, 19-9
ROLLBACK_SEGMENTS, 2-24
SHARED_MEMORY_ADDRESS, 6-13
SHARED_POOL_SIZE, 6-6, 6-12
SORT_AREA_RETAINED_SIZE, 6-15

initialization parameters *(continued)*
SORT_AREA_SIZE, 2-16, 6-15, 20-70
SORT_DIRECT_WRITES, 6-16
SQL_TRACE, 7-15
STAR_TRANSFORMATION_ENABLED, 20-79
TRANSACTIONS, 2-24
TRANSACTIONS_PER_ROLLBACK_SEGMENT, 2-24
USE_INDIRECT_DATA_BUFFERS, 6-13
initially deferred constraints, 24-20
initially immediate constraints, 24-20
INIT.ORA files, 5-4, 5-5
inner capture, 12-2
INSERT command, 14-3
direct-load INSERT, 21-2
free lists and, 2-9
parallelizing INSERT ... SELECT, 22-20
storage for parallel INSERT, 21-8
triggers and, 18-2, 18-6
BEFORE triggers, 18-8
INSTEAD OF triggers, 18-11, 18-13
INSERT privilege for object tables, 12-12, 12-13
instance groups for parallel operations, 22-16
instances, 1-6
acquire rollback segments, 2-24
associating with databases, 5-2, 5-6
defined, 1-15
described, 5-2
diagrammed, 7-6
failure in, 1-36, 28-4
instance groups, 22-16
memory structures of, 6-2
multiple-process, 7-3, 7-16
overview of, 1-6
process structure, 7-2
recovery of, 28-4
incremental checkpoints, 28-4
opening a database, 5-7
SMON process, 7-11
restricted mode, 5-5
sharing databases, 1-8
shutting down, 5-8, 5-9
single-process, 7-2
starting, 5-5
virtual memory, 6-16
INSTEAD OF triggers, 18-11
object views, 13-5
integrity constraints, 24-2
default column values and, 8-8
See also constraints
integrity rules, 1-41
parallel DML restrictions, 22-38
INTERNAL connection, 5-3
audit records not generated by, 27-5
internal errors tracked in trace files, 7-14
inter-operator parallelism, 22-11
INTERSECT operator
compound queries, 20-14
example, 20-38
optimizing view queries, 20-25
intra-operator parallelism, 22-11
INVALID status, 19-3
IS NULL predicate, 8-7
ISO SQL standard, 1-3, 10-19
composite foreign keys, 24-15
isolation levels
choosing, 23-12
read committed, 23-7
setting, 23-7, 23-30

J

job queue processes (SNP*n*), 1-19, 7-13
automatic snapshot refresh, 31-9, 31-10
message propagation, 16-8
job queues, 31-20
snapshot refresh, 31-10
JOB_QUEUE_PROCESSES parameter, 16-8
jobs, 7-2
join views, 8-13
joins
anti-joins, 20-74
Cartesian products, 20-13
cluster, 8-34, 20-47, 20-66
searches on, 20-49
convert to subqueries, 20-22
cross, 20-13
defined, 20-13
encapsulated in views, 1-43, 8-11
equijoins, 20-13

joins *(continued)*
execution plans and, 20-63
hash joins, 20-68
nested loops, 20-63
cost-based optimization, 20-69
nonequijoins, 20-13
optimization of, 20-70
outer, 20-13
non-null values for nulls, 20-72
select-project-join views, 20-24
semi-joins, 20-74
sort-merge, 20-65
cost-based optimization, 20-70
example, 20-55
views, 1-43, 8-13

K

keys
cluster, 1-45, 8-32
defined, 24-8
foreign, 24-12
hash, 8-39
in constraints, 1-55
indexes and, 8-19, 8-22, 24-9, 24-11
key values, 1-55
maximum storage for values, 8-18
parent, 24-12, 24-14
primary, 24-10
referenced, 1-55, 24-12
reverse key indexes, 8-22
searches, 20-48
unique, 24-8
composite, 24-8, 24-10

L

labels
MLSLABEL datatype, 10-16
latches
described, 23-28
LRU, 7-8
LCK*n* background processes, 7-13
See also lock processes
leaf blocks, 8-21
least recently used algorithm (LRU)
database buffers and, 6-3
dictionary cache, 4-4
full table scans and, 6-4
latches, 7-8
shared SQL pool, 6-8, 6-10
LGWR background process, 7-9
See also log writer process
LGWR_IO_SLAVES parameter, 7-11
LICENSE_MAX_SESSIONS parameter, 25-14
LICENSE_SESSIONS_WARNING parameter, 25-15
licensing
concurrent usage, 25-14
named user, 25-15
viewing current limits, 25-15
LIKE, 20-15
links, 30-6
listener processes, 7-14
literal invocation
constructor methods, 12-7
LOB datatypes, 10-9
BFILE, 10-10
BLOBs, 10-10
CLOBs and NCLOBs, 10-10
default logging mode, 21-7
local databases, 1-24
local indexes, 9-23, 9-27
bitmap indexes
on partitioned tables, 8-28
parallel query and DML, 8-24
managing partitions, 9-38
location transparency, 1-24
lock processes (LCK*n*), 1-19, 7-13
LOCK TABLE command, 14-4
LOCK_SGA parameter, 6-12, 6-16
LOCK_SGA_AREAS parameter, 6-12, 6-16
locks, 1-22, 23-3
after committing transactions, 15-6
automatic, 1-23, 23-14, 23-18
conversion, 23-16
data, 23-19
duration of, 23-15
deadlocks, 23-16, 23-17
avoiding, 23-18

locks *(continued)*
dictionary, 23-26
clusters and, 23-28
duration of, 23-28
dictionary cache, 23-29
DML acquired, 23-26
diagrammed, 23-24
DML partition locks, 9-30
escalation does not occur, 23-16
exclusive table locks (X), 23-24
file management locks, 23-29
how Oracle uses, 23-14
internal, 23-28
latches and, 23-28
log management locks, 23-29
manual, 1-23, 23-29
examples of behavior, 23-30
object level locking, 11-13
Oracle Lock Management Services, 23-40
overview of, 1-22, 23-3
parallel cache management (PCM), 23-19
parallel DML, 22-36
parse, 14-11, 23-28
rollback segment, 23-29
row (TX), 23-19
row exclusive locks (RX), 23-22
row share table locks (RS), 23-22
share row exclusive locks (SRX), 23-23
share table locks (S), 23-23
share-sub-exclusive locks (SSX), 23-23
sub-exclusive table locks (SX), 23-22
sub-share table locks (SS), 23-22
table (TM), 23-20
table lock modes, 23-21
tablespace, 23-29
types of, 23-18
log management locks, 23-29
log sequence numbers, 1-37
log writer process (LGWR), 1-18, 7-9
archiving modes, 28-16
group commits, 7-10
manual archiving and, 28-18
multiple I/O processes, 7-11
redo log buffers and, 6-6
system change numbers, 15-5
write-ahead, 7-10
LOG_ARCHIVE_START parameter, 28-17
LOG_BUFFER parameter, 6-6
system global area size and, 6-12
LOG_CHECKPOINT_INTERVAL parameter, 7-8
LOG_CHECKPOINT_TIMEOUT parameter, 7-8
LOG_FILES parameter, 6-15
logging mode
direct-load INSERT, 21-5
NOARCHIVELOG mode and, 21-5
parallel DDL, 22-25, 22-27
partitions, 9-37
SQL operations affected by, 21-7
logical blocks, 2-2
logical database structure, 1-6
logical reads limit, 25-11
logical structures, 1-8
LONG datatype
automatically the last column, 8-7
defined, 10-5
partitioning restriction, 9-12
storage of, 8-7
LONG RAW datatype, 10-11
indexing prohibited on, 10-11
partitioning restriction, 9-12
similarity to LONG datatype, 10-11
LOW_VALUE column
of USER_TAB_COLUMNS view, 20-60
LRU, 6-3, 6-4, 7-8
dictionary cache, 4-4
latches, 7-8
shared SQL pool, 6-8, 6-10

M

MAC, 1-34
mandatory access control, 1-34
manual locking, 1-23, 23-29
manual refresh, 31-10
map methods, 1-53, 11-6
massively parallel processing (MPP)
affinity, 22-6, 22-40, 22-41
multiple Oracle instances, 5-2
parallel SQL execution, 22-2
master definition site, 31-18

master group, 31-17
master site, 31-17
master table
  snapshot log, 31-8
matching foreign keys
  full, partial, or none, 24-15
MAXVALUE
  partitioned tables and indexes, 9-15
media failure, 1-36, 28-5
memory
  allocation for SQL statements, 6-10
  content of, 6-2
  cursors (statement handles), 1-15
  extended buffer cache (32-bit), 6-13
  overview of structures in, 1-13
  processes use of, 7-2
  shared SQL areas, 6-8
  software code areas, 6-16
  sort areas, 6-15
  stored procedures, 17-8, 17-15
  structures in, 6-2
  system global area (SGA)
    allocation in, 6-2
    initialization parameters, 6-12
    locking into physical memory, 6-12, 6-16
    SGA size, 6-11
    starting address, 6-13
  virtual, 6-16
MERGE hint, 20-26
MERGE_AJ hint, 20-74
MERGE_SJ hint, 20-74
merging complex views, 20-25
merging views into statements, 20-24
message queuing, 16-2
  exporting queue tables, 16-8
  messages, 16-3
  queue monitor process, 1-19, 7-13, 16-4
    interval statistics, 16-8
    window of execution, 16-5
  queue tables, 16-4
  remote databases, 16-8
methods
  comparison methods, 11-6
  constructor methods, 11-6
    literal invocation, 12-7
methods of collections
  constructor methods, 1-53
methods of object types, 1-53, 11-4
  constructor methods, 1-53, 12-4
  execution privilege for, 12-10
  map methods, 1-53, 11-6
  order methods, 1-53, 11-6
  PL/SQL, 11-12
  purchase order example, 11-2, 11-5
  selfish style of invocaton, 11-5
  use of empty parentheses with, 12-3
MINIMUM EXTENT parameter, 22-28
MINUS operator
  compound queries, 20-14
  optimizing view queries, 20-25
MLSLABEL datatype, 10-16
modes
  archive log, 28-16
  exclusive, 5-6
  shared, 5-6
  single-task, 7-16
  table lock, 23-21
  two-task, 7-16, 7-18
monitoring user actions, 1-33, 27-2
MOVE PARTITION command
  no-logging mode, 21-7
  rules of parallelism, 22-21
MPP
  See massively parallel processing
MTS_MAX_SERVERS parameter, 7-23
  artificial deadlocks and, 7-24
MTS_SERVERS parameter, 7-23
multiblock writes, 7-8
multi-master replication, 31-13
multimedia datatypes, 11-3
multiple-process systems (multiuser systems), 7-3
multiplexing
  control files, 1-38, 28-20
  recovery and, 28-6
  redo log files, 1-37
multithreaded server, 7-20
  artificial deadlocks in, 7-23
  dedicated server contrasted with, 7-20
  described, 7-16, 7-20
  dispatcher processes, 1-18, 7-13

multithreaded server *(continued)*
example of use, 7-26
Net8 or SQL*Net V2 requirement, 7-14, 7-20
processes needed for, 7-20
restricted operations in, 7-24
server processes, 1-17, 7-14, 7-23
shared server processes, 7-14, 7-23
multiuser environments, 1-2, 7-3
multiverison consistency model, 1-21
multiversion concurrency control, 23-5

N

name resolution in distributed databases, 30-6
named user licensing, 25-15
National Language Support (NLS)
character sets for, 10-4
CHECK constraints and, 24-17
clients and servers may diverge, 30-19
NCHAR and NVARCHAR2 datatypes, 10-4
NCLOB datatype, 10-10
parameters, 5-4
views and, 8-12
NCHAR datatype, 10-4
NCLOB datatype, 10-10
nested loops joins, 20-63
cost-based optimization, 20-69
nested tables, 8-9, 11-10
indexes, 12-9
Net8, 1-7, 1-26, 29-5, 30-4
Advanced Networking Option, 30-17
applications and, 29-5
client/server systems use of, 29-5
multithreaded server requirement, 7-14, 7-20
overview of, 29-5
networks
client/server architecture use of, 29-2
communication protocols, 7-27, 7-28, 29-5
dispatcher processes and, 7-14, 7-20
distributed databases, 30-2, 30-4
drivers, 7-27
failures of, 28-3
listener processes of, 7-14
Net8, 29-5, 30-4
network authentication service, 25-4
Oracle Names, 30-4
two-task mode and, 7-19
using Oracle on, 1-7, 1-26
NEXT storage parameter
parallel DML, 21-8
NLS
See National Language Support
NLS_DATE_FORMAT parameter, 10-7
NLS_LANG environment variable, 9-15
NLS_LANGUAGE parameter, 9-15
NLS_NUMERIC_CHARACTERS parameter, 10-6
NLS_SORT parameter
no effect on partitioning keys, 9-15
ORDER BY access path, 20-56
NOARCHIVELOG mode, 28-16
database backups for recovery, 28-21
defined, 28-16
LOGGING mode and, 21-5
overview, 1-38
NOAUDIT command, 14-4
locks, 23-27
nodes
disk affinity in a Parallel Server, 22-40
of distributed databases, 1-24
NOLOGGING mode
direct-load INSERT, 21-5
parallel DDL, 22-25, 22-27
partitions, 9-37
SQL operations affected by, 21-7
nonequijoins
defined, 20-13
non-prefixed indexes, 9-27
non-repeatable reads, 23-3, 23-10
non-unique indexes, 8-17
NOREVERSE option for indexes, 8-22
NOT, 20-17
NOT IN subquery, 20-74
NOT NULL constraints
constraint checking, 24-19
defined, 24-7
implied by PRIMARY KEY, 24-11
UNIQUE keys and, 24-10
novalidate constraints, 24-21
Novell NetWare Management System, 30-19

nulls
as default values, 8-8
atomic, 12-6
column order and, 8-7
converting to values, 8-7
optimization, 20-72
defined, 8-7
foreign keys and, 24-14, 24-15
how stored, 8-7
indexes and, 8-8, 8-27
inequality in UNIQUE key, 24-10
non-null values for, 8-7, 20-72
object types, 12-6
partitioned tables and indexes, 9-15
prohibited in primary keys, 24-10
prohibiting, 24-7
UNIQUE key constraints and, 24-10
unknown in comparisons, 8-7
NUM_DISTINCT column
USER_TAB_COLUMNS view, 20-60
NUM_ROWS column
USER_TABLES view, 20-60
NUMBER datatype, 10-5
internal format of, 10-7
rounding, 10-6
NVARCHAR2 datatype, 10-4
NVL function, 8-7

O

object cache
object views, 13-4
OCI, 11-13
privileges, 12-13
Pro*C, 11-13
object identifiers, 11-8, 13-3
for object types, 12-4
index on, 12-5
for object views, 13-3, 13-4
for row objects, 11-8
WITH OBJECT OID clause, 13-3, 13-4
object privileges, 26-3
See also schema object privileges
object tables, 11-3, 11-7
constraints, 12-8
indexes, 12-9
row objects, 11-8
triggers, 12-9
virtual object tables, 13-2
object types, 1-41, 11-2, 11-4
attributes of, 11-2, 11-4
column objects, 11-8
indexes, 12-9
comparison methods for, 11-6
constructor methods for, 1-53, 11-6, 12-4
incomplete, 12-14
locking in cache, 11-13
message queuing, 16-5
methods of, 1-53, 11-4
method calls, 12-3
PL/SQL, 11-12
purchase order example, 11-2, 11-5
mutually dependent, 12-13
object views, 8-14
Oracle type translator, 11-14
purchase order example, 11-2, 11-4
row objects, 11-8
use of table aliases, 12-2
object views, 8-14, 13-1
advantages of, 13-2
defining, 13-2
modifiability, 18-11
object identifiers for, 13-3, 13-4
row objects, 11-8
updating, 13-4
use of INSTEAD OF triggers with, 13-5
object-relational DBMS (ORDBMS), 1-41, 11-2
objects in a database schema, 1-6
See also schema objects
OCI, 7-27
anonymous blocks, 14-17
bind variables, 14-13
object cache, 11-13
OCIObjectFlush, 13-4
OCIObjectPin, 13-4
stored procedures, 14-18
offline backups
whole database backup, 28-21
offline redo log files, 1-37, 28-7
OIDs, 11-8, 12-4, 13-3, 13-4

WITH OBJECT OID clause, 13-3, 13-4
OLTP database, 9-4
batch jobs, 22-32
parallel DML, 22-31
partitioning indexes, 9-28
partitions, 9-5
online analytical processing (OLAP)
index-organized tables, 8-31
online redo log, 1-37, 28-7
archiving, 28-16, 28-17
checkpoints, 28-20
media failure, 28-6
multiplexed, 28-6
recorded in control file, 28-19
online transaction processing (OLTP), 9-4
reverse key indexes, 8-22
OPEN_CURSORS parameter, 14-7
managing private SQL areas, 6-9
OPEN_LINKS parameter, 6-15
operating systems
authentication by, 25-3
block size, 2-3
communications software, 7-28
privileges for administrator, 5-3
roles and, 26-16
operations in a relational database, 1-41
OPTIMAL storage parameter, 2-22
optimization, 20-2
choosing the approach, 20-40
conversion of expressions and predicates, 20-14
cost-based, 20-6, 20-69
choosing an access path, 20-58
examples of, 20-59
histograms, 20-8
remote databases and, 20-39
described, 20-2
DISTINCT, 20-25
distributed SQL statements, 20-39
execution plan for partitions, 9-10, 9-12
GROUP BY views, 20-25
hints, 20-42, 20-44
index build, 8-17
manual, 20-42
merging complex views, 20-25
merging views into statements, 20-24
non-null values for nulls, 20-72
operations performed, 20-12
parallel SQL, 22-9
partition pruning, 9-3
indexes, 9-28
partitioned indexes, 9-28
PL/SQL, 20-42
rule-based, 20-11, 20-70
choosing an access path, 20-62
examples of, 20-62
selectivity of queries and, 20-59
select-project-join views, 20-24
semi-joins, 20-74
statistics, 20-41
transitivity and, 20-17
types of SQL statements, 20-13
without merging, 20-34
OPTIMIZER_GOAL option, 20-41
OPTIMIZER_MODE, 20-40
hints affecting, 20-42
Oracle
adherence to standards, 1-3
integrity constraints, 24-5
architecture, 1-8, 1-13
client/server architecture of, 29-2
compatibility, 1-4
configurations of, 7-2, 7-16
multiple-process Oracle, 7-3, 7-16
single-process Oracle, 7-2
connectibility, 1-4
different Oracle versions, 30-7
data access, 1-48
examples of operations, 1-19, 7-24
features, 1-2
instances, 1-6, 1-15, 5-2
licensing of, 25-14
Oracle server, 1-4
Parallel Server option, 1-8
See also Parallel Server
portability, 1-4
processes of, 1-16, 7-5
scalability of, 29-4
SQL processing, 14-8
Trusted Oracle, 1-34
using on networks, 1-4, 1-26

Oracle AQ, 16-1
exporting queue tables, 16-8
message queuing, 16-2
queue monitor process, 1-19, 7-13, 16-4
interval statistics, 16-8
window of execution, 16-5
queue tables, 16-4
remote databases, 16-8
Oracle blocks, 1-10, 2-2
See also data blocks
Oracle Call Interface (OCI), 7-27
anonymous blocks, 14-17
bind variables, 14-13
object cache, 11-13
OCIObjectFlush, 13-4
OCIObjectPin, 13-4
stored procedures, 14-18
Oracle code, 7-2, 7-27
Oracle Enterprise Manager
See Enterprise Manager
Oracle Forms
object dependencies and, 19-11
PL/SQL, 14-16
Oracle Names
global directory service, 30-4
Oracle Open Gateways, 30-8
Oracle Parallel Server, 1-8
See also Parallel Server
Oracle precompilers
anonymous blocks, 14-17
bind variables, 14-13
cursors, 14-10
embedded SQL, 14-5
FIPS flagger, 14-6
stored procedures, 14-18
Oracle program interface (OPI), 7-27
Oracle Replication Manager, 31-17
Oracle Security Server, 30-17
Oracle server, 1-4
See also Oracle
Oracle type translator (OTT), 11-14
ORDBMS, 1-41, 11-2
order methods, 1-53, 11-6
ORDERED hint, 20-70
OTT, 11-14
outer joins
defined, 20-13
non-null values for nulls, 20-72

P

P code, 17-16
packages, 17-4, 17-10
advantages of, 17-13
as program units, 1-52
auditing, 27-8
dynamic SQL, 14-19
examples of, 17-10, 26-8, 26-9
executing, 14-16, 17-16
for locking, 23-40
overview of, 1-44
private, 17-14
privileges
divided by construct, 26-8
executing, 26-7, 26-8
public, 17-14
queuing, 16-4
session state and, 19-6
shared SQL areas and, 6-9
storing, 17-15
validity of, 17-16
pages, 2-2
parallel backup operations, 28-12
PARALLEL clause
parallelization rules, 22-17
parallel coordinator process, 22-6
parallel DDL
extent allocation, 22-27
parallelism types, 22-3
parallelization rules, 22-17
partitioned tables and indexes, 22-25
parallel DELETE, 22-18
parallel DML, 22-29
applications, 22-31
bitmap indexes, 8-24
degree of parallelism, 22-17, 22-19
enabling PARALLEL DML, 22-32
lock and enqueue resources, 22-36
parallelism types, 22-3
parallelization rules, 22-17

parallel DML *(continued)*
recovery, 22-34
restrictions, 22-37
transaction model, 22-33
PARALLEL hint, 22-13
parallelization rules, 22-17
UPDATE and DELETE, 22-18
parallel mode, 5-6
parallel propagation, 31-21
parallel query, 22-2
bitmap indexes, 8-24
full table scans, 22-5
inter-operator parallelism, 22-11
intra-operator parallelism, 22-11
parallelization rules, 22-17
partitioned tables and indexes, 22-4
parallel recovery, 28-12, 28-13
Parallel Server, 1-8
concurrency limits and, 25-15
databases and instances, 5-2
disk affinity, 22-40
distributed locks, 23-19
DML locks and performance, 9-31
exclusive mode, 5-6
rollback segments and, 2-24
file and log management locks, 23-29
instance groups, 22-16
isolation levels, 23-11
lock processes, 1-19, 7-13
mounting a database using, 5-6
named user licensing and, 25-16
parallel SQL, 22-1
PCM locks, 23-19
reverse key indexes, 8-22
shared mode, 5-6
rollback segments and, 2-24
system change numbers, 7-10
system monitor process and, 7-11
parallel server process, 22-6
parallel SQL
allocating rows to parallel server processes, 22-9
coordinator process, 22-6
degree of parallelism, 22-13
instance groups, 22-16
multithreaded server, 22-8
number of parallel server processes, 22-7
operations in execution plan, 22-9
optimizer, 22-9
Parallel Server and, 22-1
parallelization rules, 22-17
summary or rollup tables, 22-26
parallel UPDATE, 22-18
PARALLEL_DEFAULT_MAX_SCANS parameter (obsolete), 22-15
PARALLEL_DEFAULT_SCANSIZE parameter (obsolete), 22-15
PARALLEL_INDEX hint, 22-13
PARALLEL_MAX_SERVERS parameter, 22-8
PARALLEL_MIN_PERCENT parameter, 22-15
PARALLEL_MIN_SERVERS parameter, 22-7, 22-8
PARALLEL_SERVER_IDLE_TIME parameter, 22-8
parameter files, 5-4
example of, 5-4
used at startup, 5-5
parameters
initialization, 5-4
locking behavior, 23-18
See also initialization parameters
National Language Support, 5-4
storage, 2-5, 2-11
parentheses, use of in method calls, 12-3
parse trees, 17-15
construction of, 14-7
in shared SQL area, 6-8
stored in database, 17-16
parsing, 14-11
DBMS_SQL package, 14-19
embedded SQL, 14-6
parse calls, 14-8
parse locks, 14-11, 23-28
performed, 14-8
SQL statements, 14-11, 14-19
partial backups, 28-22
partition views, 9-10
partitioning columns, 9-12
partitioning keys, 9-12, 9-15
multi-column keys, 9-16

partitions, 9-2, 9-11
advantages of, 9-4, 9-6
affinity, 22-40
basic partitioning model, 9-11
bitmap indexes, 8-28
concurrent maintenance operations, 9-33
DML partition locks, 9-30
dynamic partitioning, 22-6
equipartitioning, 9-18
EXCHANGE PARTITION, 9-10
execution plan, 9-10, 9-12
global indexes, 9-25, 9-39
local indexes, 9-23, 9-38
LONG and LONG RAW restriction, 9-12
maintenance operations, 9-31
no-logging mode, 21-7
OLTP databases, 9-5
parallel DDL, 22-25
parallel queries, 22-4
partition bounds, 9-14
partition names, 9-14
partition pruning, 9-3
disk striping and, 22-41
indexes, 9-28
parallelizing by block range, 22-4
TO_DATE format mask, 9-12, 9-16
partition transparency, 9-9
partition-extended table names, 9-42
partitioning indexes, 9-22, 9-28
partitioning keys, 9-12, 9-15
partitioning tables, 9-21
physical attributes, 9-21, 9-29
prefixed indexes, 9-24
range partitioning, 9-12
disk striping and, 22-41
rebuild partition, 9-39
referencing a partition, 9-14
restrictions
bitmap indexes, 9-12
datatypes, 9-12, 9-16
partition-extended table names, 9-42
rules of parallelism, 22-21, 22-23
statistics, 9-12
VLDB, 9-4
passwords
account locking, 25-5
administrator privileges, 5-3
complexity verification, 25-6
connecting with, 7-5
connecting without, 25-3
database user authentication, 25-4
encryption, 25-4
expiration, 25-5
password files, 25-7
password reuse, 25-5
used in roles, 1-31
PCTFREE storage parameter
how it works, 2-6
PCTUSED and, 2-8
PCTINCREASE storage parameter
parallel DML, 21-9
PCTUSED storage parameter
how it works, 2-6
PCTFREE and, 2-8
performance
clusters and, 8-34
constraint effects on, 24-6
DSS database, 9-7, 22-31
dynamic performance tables (V$), 4-7
group commits, 7-10
I/O, 9-8
index build, 8-17
Oracle Parallel Server and DML locks, 9-31
packages, 17-14
parallel recovery and, 28-14
partitions, 9-7
prefixed and non-prefixed indexes, 9-28
recovery, 28-5
resource limits and, 25-10
SGA size and, 6-11
structures that improve, 1-45
viewing execution plans, 20-4
persistent areas, 6-8
persistent queuing, 16-2
PGA, 1-16, 6-13
multithreaded server, 7-23
phantom reads, 23-3, 23-10
physical database structure, 1-5

PL/SQL, 1-52, 14-15
anonymous blocks, 14-15, 17-9
auditing of statements within, 27-4
bind variables
user-defined types, 11-12
database triggers, 1-55, 18-1
datatypes, 10-2
dynamic SQL, 14-19
exception handling, 14-18
executing, 14-15, 17-16, 17-17
external procedures, 14-19, 17-9
language constructs, 14-17
object views, 13-4
optimizer goal, 20-42
packages, 17-4, 17-10
parse locks, 23-28
parsing DDL statements, 14-19
PL/SQL engine, 14-15, 17-2
compiler, 17-15
executing a procedure, 17-17
products containing, 14-16
program units, 1-44, 6-9, 14-15, 17-2
compiled, 14-16, 17-8, 17-15
shared SQL areas and, 6-9
roles disabled in named PL/SQL blocks, 26-14
stored procedures, 1-44, 14-15, 17-2, 17-6
user locks, 23-40
user-defined datatypes, 11-12
plan
accessing views, 20-28, 20-31, 20-32
complex statements, 20-23
compound queries, 20-36, 20-37, 20-38
joining views, 20-34
joins, 20-63, 20-69
OR operators, 20-20
SQL execution, 14-4, 14-11
star transformation, 20-78
PMON background process, 7-12
See also process monitor process
portability, 1-4
precompilers
anonymous blocks, 14-17
bind variables, 14-13
cursors, 14-10
embedded SQL, 14-5
FIPS flagger, 14-6
stored procedures, 14-18
predicates
optimizing view queries, 20-24
partition pruning, 9-3
indexes, 9-28
pushing into a view, 20-27, 20-32
examples, 20-28, 20-30
prefixed indexes, 9-24, 9-27
prefixes of data dictionary views, 4-5
PRIMARY KEY constraints, 24-10
constraint checking, 24-19
described, 24-10
indexes used to enforce, 24-11
name of, 24-11
maximum number of columns, 24-11
NOT NULL constraints implied by, 24-11
primary key snapshot, 31-11
primary keys, 1-55, 24-10
advantages of, 24-10
defined, 24-3
optimization, 20-23
searches, 20-48
primary ownership, 31-23
private rollback segments, 2-23
private SQL areas
cursors and, 6-9
described, 6-8
how managed, 6-9
persistent areas, 6-8
runtime areas, 6-8
privileges
administrator
not audited, 27-5
auditing use of, 1-33, 27-7
checked when parsing, 14-11
granting, 1-30, 26-3, 26-4
examples of, 26-8, 26-9
grouping into roles, 1-30
overview of, 1-30, 26-2
partitioned tables and indexes, 9-41
procedures, 26-7
creating and altering, 26-8
executing, 17-16, 26-7
in packages, 26-8

privileges *(continued)*
RESTRICTED SESSION, 25-15
revoked
object dependencies and, 19-6
revoking, 26-3, 26-4
roles, 26-10
restrictions on, 26-15
schema object, 26-3
DML and DDL operations, 26-4
granting and revoking, 26-4
overview of, 1-30
packages, 26-8
procedures, 26-7
system, 26-2
granting and revoking, 26-3
overview of, 1-30
user-defined types, 12-10
to start up or shut down a database, 5-3
trigger privileges, 26-8
user-defined types
acquired by role, 12-10
ALTER ANY TYPE, 12-10
checked when pinning, 12-13
column level for object tables, 12-13
CREATE ANY TYPE, 12-10
CREATE TYPE, 12-10
DELETE, 12-12, 12-13
DROP ANY TYPE, 12-10
EXECUTE, 12-10, 12-11, 12-12, 12-13
EXECUTE ANY TYPE, 12-10, 12-11
EXECUTE ANY TYPE with ADMIN OPTION, 12-11
EXECUTE with GRANT option, 12-11
INSERT, 12-12, 12-13
SELECT, 12-12, 12-13
system privileges, 12-10
UPDATE, 12-12, 12-13
using, 12-11, 12-15
views, 26-6
creating, 26-6
using, 26-6
Pro*C / C++
processing SQL statements, 14-10
user-defined datatypes, 11-12
procedural replication, 31-26
detecting conflicts, 31-26
wrapper, 31-26
procedures, 14-15, 17-1, 17-6, 19-7
advantages of, 17-7
auditing, 27-8
contrasted with anonymous blocks, 17-8
contrasted with functions, 1-52, 17-2
cursors and, 14-17
dependency tracking in, 19-6
examples of, 17-6, 26-8, 26-9
executing, 14-16, 17-16
external procedures, 14-19, 17-9
INVALID status, 19-3, 19-6
prerequisites for compilation of, 19-5
privileges
create or alter, 26-8
executing, 26-7
executing in packages, 26-8
remote procedure calls, 30-11
roles disabled in, 26-14
security enhanced by, 17-7, 26-7
shared SQL areas and, 6-9
stored procedures, 14-15, 14-18, 17-2
storing, 17-15
triggers, 18-2
validity of, 17-16
process global area (PGA), 6-13
See also program global area
process monitor process (PMON)
cleans up timed-out sessions, 25-12
described, 1-18, 7-12
network failure, 28-3
parallel DML process recovery, 22-34
process failure, 28-3
processes, 7-2
archiver (ARCH), 1-18, 7-12
background, 1-17, 7-6
diagrammed, 7-6
checkpoint (CKPT), 1-18, 7-11
checkpoints and, 7-8
database writer (DBW*n*), 1-17, 7-8
dedicated server, 7-23
dispatcher (D*nnn*), 1-18, 7-13

processes *(continued)*
distributed transaction resolution, 7-12
during recovery, 28-14
failure in, 28-3
job queue (SNP*n*), 1-19, 7-13
message propagation, 16-8
snapshot refresh, 31-9, 31-10
listener, 7-14
shared servers and, 7-20
lock (LCK*n*), 1-19, 7-13
log writer (LGWR), 1-18, 7-9
multiple-process Oracle, 7-3
multithreaded server, 7-20
artificial deadlocks and, 7-23
client requests and, 7-21
Oracle, 1-16, 7-5
single-process Oracle, 7-2
overview of, 1-16
parallel coordinator, 22-6
parallel server processes, 22-6
process monitor (PMON), 1-18, 7-12
queue monitor (QMN*n*), 1-19, 7-13, 16-4
recoverer (RECO), 1-18, 7-12
and in-doubt transactions, 1-25
server, 1-16, 1-24, 7-5
dedicated, 7-18
shared, 7-13, 7-14, 7-23
shadow, 7-18
structure, 7-2
system monitor (SMON), 1-18, 7-11
trace files for, 7-14
user, 1-16, 7-4
allocate PGAs, 6-13
recovery from failure of, 7-12
sharing server processes, 7-13, 7-14
processing
DDL statements, 14-14
distributed, 1-23
DML statements, 14-10
overview, 14-8
parallel SQL, 22-2
queries, 14-11
profiles
overview of, 1-32
password management, 25-5
when to use, 25-13
program global area (PGA), 1-16, 6-13
allocation of, 6-13
contents of, 6-13
multithreaded servers, 7-23
nonshared and writable, 6-13
size of, 6-14
program interface, 7-27
Oracle side (OPI), 7-27
overview of, 1-19
single-task mode in, 7-17
structure of, 7-27
two-task mode in, 7-19
user side (UPI), 7-27
program units, 1-44, 14-15, 17-2
prerequisites for compilation of, 19-5
shared pool and, 6-9
propagation
parallel, 31-21
serial, 31-21
propagator of replicated data, 31-22
pruning partitions, 9-3, 22-4, 22-41
index partitions, 9-4
indexes, 9-28
TO_DATE format mask, 9-12, 9-16
pseudocode, 17-16
triggers, 18-17
pseudocolumns
CHECK constraints prohibit
LEVEL and ROWNUM, 24-16
modifying views, 18-12
ROWID, 10-12
ROWNUM
cannot use indexes, 20-58
optimizing view queries, 20-25, 20-34
USER, 26-7
public rollback segments, 2-23
PUBLIC user group, 25-9, 26-14
validity of procedures, 17-16
purchase order example
object types, 11-2, 11-4
PUSH_JOIN_PRED hint, 20-73
PUSH_JOIN_PREDICATE parameter, 20-73

Q

QMN*n* background process, 1-19, 7-13, 16-4
  interval statistics, 16-8
  window of execution, 16-5
queries
  ad hoc, 22-26
  compound
    defined, 20-14
    optimization of, 20-36
    ORs converted to, 20-19
  default locking of, 23-25
  define phase, 14-12
  defined, 20-13
  describe phase, 14-12
  distributed or remote, 30-10
  fetching rows, 14-11
  in DML, 14-3
  index scans parallelized by partition, 22-5
  location transparency and, 30-14
  merged with view queries, 8-12
  optimizing IN subquery, 20-25
  optimizing view queries, 20-24
  parallel processing, 22-2
  phases of, 23-5
  processing, 14-11
  read consistency of, 1-22, 23-5
  selectivity of, 20-59
  star queries, 20-75
  stored as views, 1-43, 8-10
  table scans parallelized by rowid, 22-4
  temporary segments and, 2-16, 14-12
  triggers use of, 18-16
queue monitor process (QMN*n*), 1-19, 7-13, 16-4
  interval statistics, 16-8
  window of execution, 16-5
queuing, 16-2
  exporting queue tables, 16-8
  queue monitor process, 1-19, 7-13, 16-4
    interval statistics, 16-8
    window of execution, 16-5
  queue tables, 16-4, 16-8
  remote databases, 16-8
quotas
  revoking tablespace access and, 25-9
  setting to zero, 25-9
  tablespace, 1-32, 25-8
    temporary segments ignore, 25-9

R

range partitioning, 9-12
RAW datatype, 10-11
RDBMS, 1-41
  object-relational DBMS, 1-41, 11-2
  See also Oracle
read committed isolation, 23-6, 23-7
read consistency
  defined, 1-21
  multiversion consistency model, 1-21
  queries, 14-12
  rollback segments and, 2-18
  snapshot too old message, 23-5
  subqueries in DML, 23-13
  transactions, 1-21, 23-6
  triggers and, 18-14, 18-16
read snapshot time, 23-10
read uncommitted, 23-3
readers block writers, 23-10
read-only replication, 31-2, 31-4
  uses of, 31-4
read-only snapshot, 31-6
  refresh types, 31-9
read-only tablespaces
  backing up, 28-23
  described, 3-9
  restrictions on, 3-10
read-only transactions, 1-22
reads
  data block
    limits on, 25-11
  dirty, 23-2
  repeatable, 23-6
real-time replication, 31-27
REBUILD INDEX command
  no-logging mode, 21-7
  rules of parallelism, 22-21
REBUILD INDEX PARTITION command, 9-39
  no-logging mode, 21-7
  rules of parallelism, 22-21

receiver of replicated data, 31-22
recoverer process (RECO), 1-18, 7-12
in-doubt transactions, 1-25, 5-7, 15-8
recovery
basic steps, 1-39, 28-9
database buffers and, 28-8
diagrammed, 28-15
disaster recovery, 28-24
distributed processing in, 7-12
instance
SMON process, 7-11
instance recovery, 28-4
incremental checkpoint, 28-5
opening a database, 5-7
parallel DML, 22-35
required after abort, 5-8
media recovery
dispatcher processes, 7-24
enabled or disabled, 28-16
of distributed transactions, 5-7
overview of, 1-34, 28-8
parallel DML, 22-34
parallel recovery, 28-13
parallel restore, 28-12
process recovery, 7-12, 28-3
recommendations for, 28-15
Recovery Manager, 1-40, 28-10
rolling back during, 28-9
rolling forward and, 28-9
standby database, 28-24
statement failure, 28-3
structures used in, 1-37, 28-7
whole database backups, 28-21
Recovery Manager, 1-40, 28-10
generating reports, 28-12
operating without a catalog, 28-11
parallel operations, 28-12
recovery catalog, 28-10
recursive SQL
cursors and, 14-7
redo log buffers, 1-15, 6-6
circularity, 7-9
committing a transaction, 7-10
log writer process and, 6-6
size of, 6-6
writing, 7-9
redo log files, 1-12, 28-7
archived, 1-37, 28-16
automatically, 28-17
errors in archiving, 28-18
manually, 28-18
archiver process (ARCH), 7-12
buffer management, 7-9
files named in control file, 28-19
log sequence numbers, 1-37
recorded in control file, 28-19
log writer process, 7-9
mode of, 1-38
multiplexed, 1-37
purpose of, 1-12
online or offline, 1-37, 28-7
overview of, 1-12, 1-37
parallel recovery, 28-13
physical database structure, 1-5
recovery and, 28-7
rolling forward and, 28-9
when temporary segments in, 2-17
written before transaction commit, 7-10
REF targets, 12-14
See also REFs
referenced
keys, 1-55, 24-12
objects, 19-2
partitions, 9-14
REFERENCES privilege
when granted through a role, 26-15
referential integrity, 23-11, 24-12
cascade rule, 24-3
examples of, 24-17
partially null foreign keys, 24-15
PRIMARY KEY constraints, 24-10
restrict rule, 24-3
self-referential constraints, 24-14, 24-17
set to default rule, 24-3
set to null rule, 24-3
refresh
job queue processes (SNP*n*), 1-19, 7-13
refresh group, 31-9
automatic refresh, 31-9
manual refresh, 31-10

refresh interval
snapshot refresh group, 31-9
refresh types
read-only snapshots, 31-9
REFs, 11-8
constructing from object identifiers, 12-4, 12-5
dangling, 11-8, 11-9
dereferencing of, 11-9
for rows of object views, 13-3
implicit dereferencing of, 11-9
indexes on, 12-9
mutually dependent types, 12-13
pinning, 12-13, 13-4
potential REF targets
REF targets, 12-14
scoped, 11-8, 12-5
size of, 12-5
use of table aliases, 12-2
relational DBMS (RDBMS)
object-relational DBMS, 11-2
principles, 1-40
SQL and, 14-2
See also Oracle
relations, 1-42
remote databases, 1-24, 30-2
database links, 30-6
See also distributed databases
remote dependencies, 19-9
remote procedure calls, 30-11, 31-19
remote transactions, 30-11
REMOTE_DEPENDENCIES_MODE parameter, 19-9
RENAME command, 14-4
repeatable reads, 23-3
replication, 31-1
administrator, 31-22
basic, 31-2, 31-4
catalog, 31-18
conflicts, 31-22
column groups, 31-25
data models and, 31-23
detecting, 31-24
procedural replication, 31-26
resolution methods, 31-25
resolving, 31-25
row-level replication, 31-24
definition, 31-2
distributed databases vs., 30-7
group, 31-17
object, 31-17
procedural, 31-26
propagator, 31-22
real-time, 31-27
receiver, 31-22
restrictions
direct-load INSERT, 21-10
parallel DML, 22-38
site, 31-17
symmetric, 31-11, 31-12
uses of read-only, 31-4
uses of symmetric, 31-12
replication management API, 31-18
Replication Manager, 31-17
reserved words, 14-3
resource limits
call level, 25-11
connect time per session, 25-12
CPU time limit, 25-11
determining values for, 25-13
idle time per session, 25-12
logical reads limit, 25-11
overview of, 1-32
private SGA space per session, 25-12
session level, 25-10
sessions per user, 25-12
RESOURCE role, 26-16
user-defined types, 12-10, 12-11
response queues, 7-21
response time, 20-7
cost-based approach, 20-40
restricted mode
starting instances in, 5-5
restricted ROWID format, 10-13
RESTRICTED SESSION privilege, 25-15
restrictions
direct-load INSERT, 21-9
parallel DML and direct-load INSERT, 22-37
partition views, 9-11
partitions
bitmap indexes, 9-12

restrictions *(continued)*
datatypes, 9-12, 9-16
partition-extended table names, 9-42
reverse key indexes, 8-22
REVERSE option for indexes, 8-22
REVOKE command, 14-4
FORCE option, 12-15
locks, 23-27
object types and dependencies, 12-15
roles, 1-31, 26-10
application, 26-12
CONNECT role, 12-10, 12-11, 26-16
DBA role, 12-10, 26-16
DDL statements and, 26-14
dependency management in, 26-15
disabled in named PL/SQL blocks, 26-14
distributed database applications, 30-16
enabled or disabled, 26-13
EXP_FULL_DATABASE role, 26-16
functionality, 26-2
global authentication service, 30-16
granting, 26-3, 26-13
IMP_FULL_DATABASE role, 26-16
in applications, 1-31
managing via operating system, 26-16
naming, 26-14
overview of, 1-31
predefined, 26-16
queue administrator, 16-5
RESOURCE role, 12-10, 12-11, 26-16
restrictions on privileges of, 26-15
revoking, 26-13
schemas do not contain, 26-14
security domains of, 26-14
use of passwords with, 1-31
user, 26-12
users capable of granting, 26-13
uses of, 26-11
rollback, 2-17, 15-6
defined, 1-50
described, 15-6
during recovery, 1-40, 28-9
ending a transaction, 15-2, 15-4, 15-6
fast transaction rollback, 28-10
statement-level, 15-4
to a savepoint, 15-6
ROLLBACK command, 14-5
rollback entries, 2-17
rollback segments, 1-11, 2-17
access to, 2-17
acquired during startup, 5-7
allocation of extents for, 2-19
new extents, 2-21
clashes when acquiring, 2-24
committing transactions and, 2-19
contention for, 2-19
deallocating extents from, 2-22
deferred, 2-27
defined, 1-11
dropping, 2-22
restrictions on, 2-27
how transactions write to, 2-19
in-doubt distributed transactions, 2-21
invalid, 2-25
locks on, 23-29
moving to the next extent of, 2-20
number of transactions per, 2-19
offline, 2-25, 2-27
offline tablespaces and, 2-27
online, 2-25, 2-27
overview of, 2-17, 28-8
parallel recovery, 28-10
partly available, 2-25, 28-4
private, 2-23
public, 2-23
read consistency and, 1-21, 2-18, 23-4
recovery needed for, 2-25
states of, 2-25
SYSTEM rollback segment, 2-23
transactions and, 2-18
use of in recovery, 1-38, 28-9
when acquired, 2-23
when used, 2-18
written circularly, 2-19
rolling back during recovery, 28-9
rolling back transactions, 1-51, 15-2, 15-6
fast warmstart, 28-4
rolling forward during recovery, 1-39, 28-9
root blocks, 8-41
row cache, 6-10

row data (section of data block), 2-5
row directories, 2-4
row locking, 23-10, 23-19
  serializable transactions and, 23-8
row objects, 11-8
row pieces, 8-5
  headers, 8-5
  how identified, 8-7
row sources, 20-3
row triggers, 18-7, 18-8
  when fired, 18-14
  See also triggers
ROWID datatype, 10-12
  extended ROWID format, 10-12
  restricted ROWID format, 10-13
ROWID snapshot, 31-11
ROWIDs, 8-7
  accessing, 10-12
  changes in, 10-12
  in non-Oracle databases, 10-15
  in REFs, 12-5
  internal use of, 10-15
  of clustered rows, 8-7
  sorting indexes by, 8-21
  table access by, 20-43
ROWLABEL column, 10-16
row-level locking, 23-10, 23-19
row-level replication, 31-19
  detecting conflicts, 31-24
ROWNUM pseudocolumn
  cannot use indexes, 20-58
  optimizing view queries, 20-25, 20-34
rows, 1-42, 8-3
  addresses of, 8-7
  chaining across blocks, 2-10, 8-5
  clustered, 8-6
    ROWIDs of, 8-7
  defined, 1-42
  described, 8-3
  fetched, 14-11
  format of in data blocks, 2-4
  headers, 8-5
  locking, 23-10, 23-19
  locks on, 9-30, 23-19, 23-22
  pieces of, 8-5
  row objects, 11-8
  row overflow in index-organized tables, 8-29
  row sources, 20-3
  ROWIDs used to locate, 20-43, 20-47
  shown in ROWIDs, 10-13, 10-14
  size of, 8-5
  storage format of, 8-5
  triggers on, 18-8
  when ROWID changes, 10-12
RPC, 30-11, 31-19
RULE hint
  OPTIMIZER_MODE and, 20-42
rule-based optimization, 20-11
runtime areas, 6-8

S

same-row writers block writers, 23-10
SAVEPOINT command, 14-5
savepoints, 1-51, 15-7
  described, 15-7
  implicit, 15-4
  overview of, 1-51
  rolling back to, 15-6
scalability
  batch jobs, 22-32
  client/server architecture, 29-4
  parallel DML, 22-31
  parallel SQL execution, 22-2
scans, 20-43
  cluster, 20-47, 20-48, 20-49, 20-50
    indexed, 20-50
  fast full index scan, 20-44
  full table, 20-43, 20-57
    LRU algorithm, 6-4
    multiblock reads, 20-59
    parallel query, 22-5
    rule-based optimizer, 20-62
  hash cluster, 20-48, 20-50
  index, 20-43
    bitmap, 20-45
    bounded range, 20-53
    cluster key, 20-50
    composite, 20-51
    MAX or MIN, 20-55

scans, index *(continued)*
ORDER BY, 20-56
restrictions, 20-57
selectivity and, 20-59
single-column, 20-51
unbounded range, 20-54
range, 20-44, 20-51
bounded, 20-53
MAX or MIN, 20-55
ORDER BY, 20-56
unbounded, 20-54
table scan and CACHE clause, 6-4
unique, 20-44, 20-48, 20-50
schema names
in distributed databases, 30-6
qualifying column names, 12-3
unique within a database, 30-6
schema object privileges, 26-3
DML and DDL operations, 26-4
granting and revoking, 26-4
overview of, 1-30
views, 26-6
schema objects, 8-1
auditing, 1-33, 27-8
creating
tablespace quota required, 25-8
default tablespace for, 25-8
defined, 1-6
dependencies of, 19-2
and distributed databases, 19-11
and views, 8-13
on non-existence of other objects, 19-7
triggers manage, 18-14
dependent on lost privileges, 19-6
global names, 30-6
in a revoked tablespace, 25-9
information about, 4-2
INVALID status, 19-3
names in distributed databases, 30-6
overview of, 1-10, 1-42, 8-2
privileges on, 26-3
relationship to datafiles, 3-12, 8-2
trigger dependencies on, 18-18
user-defined types, 11-3
schemas, 25-2
associated with users, 1-28, 8-2
contents of, 8-2
contrasted with tablespaces, 8-2
defined, 25-2
objects in, 8-2
user-defined datatypes, 11-12
SCN, 15-5
See also system change numbers
scoped REFs, 11-8, 12-5
security, 1-31, 25-2
administrator privileges, 5-3
application enforcement of, 1-31
auditing, 27-2, 27-6
auditing user actions, 1-33
data, 1-28
data encryption, 30-17
deleting audit data, 4-5
described, 1-27
discretionary access control, 1-28, 25-2
distributed databases, 30-16
domains, 1-29, 25-2
enforcement mechanisms, 1-28
message queues, 16-5
passwords, 25-4
procedures enhance, 26-7
program interface enforcement of, 7-27
system, 1-28, 4-3
views and, 8-11
views enhance, 26-6
security domains, 1-29, 25-2
enabled roles and, 26-13
tablespace quotas, 25-8
segments, 1-10, 2-15
data, 2-15
deallocating extents from, 2-13
defined, 2-3
header block, 2-11
index, 2-15
overview of, 1-10, 2-15
rollback, 2-17
temporary, 1-11, 2-16
allocating, 2-16
cleaned up by SMON, 7-11

segments, temporary *(continued)*
dropping, 2-14
ignore quotas, 25-9
operations that require, 2-16
parallel INSERT, 21-8
tablespace containing, 2-14, 2-16
SELECT command, 14-3
subqueries, 14-12
See also queries
SELECT privilege for object tables, 12-12, 12-13
selectivity of queries, 20-59
select-project-join views, 20-24
selfish style of method invocation, 11-5
semi-joins, 20-74
sequences, 1-43, 8-15
auditing, 27-8
CHECK constraints prohibit, 24-16
coordinated generation, 31-24
independence from tables, 8-15
length of numbers, 8-15
number generation, 8-14
serial propagation, 31-21
Server Manager
ALERT file, 7-15
executing a package, 17-6
executing a procedure, 17-4
lock and latch monitors, 23-28
PL/SQL, 14-17, 14-18
session variables, 14-17
showing size of SGA, 6-12
SQL statements, 14-2
statistics monitor, 25-13
server processes, 1-16, 7-5
servers, 1-23
client/server architecture, 29-2
dedicated, 1-17, 7-18
multithreaded contrasted with, 7-20
dedicated server architecture, 7-16
defined, 1-24
multithreaded, 1-17
architecture, 7-16, 7-20
dedicated contrasted with, 7-20
processes of, 7-13, 7-14, 7-20, 7-23
processes of, 1-16
shared, 1-17
session control statements, 1-49, 14-5
SESSION_ROLES view
queried from PL/SQL block, 26-14
sessions
auditing by, 27-10
connections contrasted with, 7-4
defined, 7-5, 27-10
enabling PARALLEL DML, 22-32
limit on concurrent, 1-32
by license, 25-14
limits per user, 25-12
package state and, 19-6
resource limits and, 25-10
stack space in PGA, 6-13
time limits on, 25-12
transaction isolation level, 23-30
when auditing options take effect, 27-6
where information is stored, 6-13
SET CONSTRAINTS command
DEFERRABLE or IMMEDIATE, 24-20
SET ROLE command, 14-5
SET TRANSACTION command, 14-5
ISOLATION LEVEL, 23-7, 23-30
READ ONLY, 2-18
SGA
See system global area
shadow column group, 31-25
shadow processes, 7-18
share locks
share table locks (S), 23-23
shared global area (SGA), 6-2
See also system global area
shared mode, 5-6
rollback segments, 2-24
shared ownership, 31-24
shared pool, 6-6
allocation of, 6-10
ANALYZE command and, 6-11
dependency management and, 6-11
flushing, 6-11
object dependencies and, 19-8
overview of, 1-15
procedures and packages, 17-15
row cache and, 6-10
size of, 6-6

shared server processes (S*nnn*), 7-14, 7-23
described, 7-23
shared servers, 1-17
cannot connect with administrator privileges, 5-3
shared SQL areas, 6-8, 14-7
ANALYZE command and, 6-11
dependency management and, 6-11
described, 6-8
loading SQL into, 14-11
overview of, 1-15, 14-7
parse locks and, 23-28
procedures, packages, triggers and, 6-9
size of, 6-8
SHARED_MEMORY_ADDRESS parameter, 6-13
SHARED_POOL_SIZE parameter, 6-6
system global area size and, 6-12
shutdown, 5-8, 5-9
abnormal, 5-6, 5-9
deallocation of the SGA, 6-2
prohibited by dispatcher processes, 7-24
steps, 5-8
SHUTDOWN ABORT command, 5-9
signature checking, 19-9
Simple Network Management Protocol (SNMP) support
database management, 30-19
simple snapshot
structure, 31-7
single-process systems (single-user systems), 7-2
single-task mode, 7-16
site autonomy, 1-25, 30-15
skewing parallel DML workload, 22-16
SMON background process, 7-11
See also system monitor process
SMP architecture
disk affinity, 22-41
snapshot
complex, 31-10
defining query, 31-7
group, 31-18
log, 31-8
primary key, 31-11
read-only, 31-6
refresh, 7-13, 31-8, 31-10
rowid, 31-11
simple
structure, 31-7
site, 31-18
subquery, 31-7
updatable, 31-14
updating, 31-8
snapshot refresh
complete, 31-8
fast, 31-8
group
refresh interval, 31-9
groups, 31-9
job queue processes (SNP*n*), 1-19, 7-13
automatic snapshot refresh, 31-9
snapshot log and, 31-8
snapshot too old message, 23-5
S*nnn* background processes, 7-14
SNP*n* background processes, 1-19, 7-13
for automatic snapshot refresh, 31-9, 31-10
message propagation, 16-8
software code areas, 6-16
shared by programs and utilities, 6-17
SOME, 20-15
sort areas, 6-15
sort direct writes feature, 6-16
SORT_AREA_RETAINED_SIZE parameter, 6-15
SORT_AREA_SIZE parameter, 2-16, 6-15
cost-based optimization and, 20-70
SORT_DIRECT_WRITES parameter, 6-16
sort-merge joins, 20-65
access path, 20-55
cost-based optimization, 20-70
example, 20-55
space management
compression of free space, 2-9
direct-load INSERT, 21-8
MINIMUM EXTENT parameter, 22-28
parallel DDL, 22-27
PCTFREE, 2-6
PCTUSED, 2-6
row chaining, 2-10
segments, 2-15
spatial applications
index-organized tables, 8-31

SPLIT PARTITION command
no-logging mode, 21-7
rules of parallelism, 22-21
SQL, 14-2
cursors used in, 14-6
Data Definition Language (DDL), 14-4
Data Manipulation Language (DML), 14-3
dynamic SQL, 14-19
embedded, 1-49, 14-5
user-defined datatypes, 11-13
functions, 14-2
column default values, 8-8
COUNT, 8-27
in CHECK constraints, 24-16
NVL, 8-7
optimizing view queries, 20-25, 20-32
memory allocation for, 6-10
overview of, 1-48, 14-2
parallel execution, 22-2
parsing of, 14-7
PL/SQL and, 1-52, 14-15
recursive, 14-6
cursors and, 14-7
reserved words, 14-3
session control statements, 14-5
shared SQL, 14-7
statement-level rollback, 15-4
system control statements, 14-5
transaction control statements, 14-5
transactions and, 1-49, 15-2, 15-5
types of statements in, 1-48, 14-3
optimizing, 20-13
user-defined datatypes, 11-12, 12-2
embedded SQL, 11-13
OCI, 11-13
SQL areas
private, 6-8
persistent, 6-8
runtime, 6-8
shared, 1-15, 6-8, 14-7
SQL statements, 1-48, 14-3, 14-8
array processing, 14-13
auditing, 27-7, 27-9
overview, 1-33
when records generated, 27-4
complex, 20-13, 20-22
optimizing, 20-22
converting
examples of, 20-19
creating cursors, 14-10
dictionary cache locks and, 23-29
distributed
defined, 20-14
optimization of, 20-39
routing to nodes, 14-11
distributed databases and, 30-10
embedded, 14-5
execution, 14-8, 14-13
execution plans of, 20-2
failure in, 28-2
handles, 1-15
number of triggers fired by single, 18-14
optimization
complex statements, 20-22
types of statements, 20-13
parallel query, 22-2
parallelizing, 22-2, 22-9
parse locks, 23-28
parsing, 14-11
privileges required for, 26-3
recursive
OPTIMIZER_GOAL does not affect, 20-41
referencing dependent objects, 19-4
resource limits and, 25-11
simple, 20-13
successful execution, 15-3
transactions, 14-14
triggers on, 18-2, 18-8
triggering events, 18-6
types of, 1-48, 14-3, 20-13
SQL*Loader
Direct Loader, 21-2
partition operations, 9-31, 9-33
SQL*Menu
PL/SQL, 14-16
SQL*Module
FIPS flagger, 14-6
stored procedures, 14-18
SQL*Net
See Net8

SQL*Plus
anonymous blocks, 14-17
connecting with, 25-3
SQL statements, 14-2
stored procedures, 14-18
SQL_TRACE parameter, 7-15
SQL92, 23-2
stack space, 6-13
standards, 1-3
ANSI/ISO, 1-3, 24-5, 24-15
isolation levels, 23-2, 23-10
FIPS, 1-3, 14-6
integrity constraints, 24-5, 24-15
Oracle adherence, 1-3
standby databases, 28-24
STAR hint, 20-76
star query, 20-76
extended star schemas, 20-76
hints, 20-76
indexes, 20-76
tuning, 20-75
star transformation, 20-76
example, 20-77
restrictions, 20-80
STAR_TRANSFORMATION hint, 20-79
STAR_TRANSFORMATION_ENABLED parameter, 20-79
startup, 5-2, 5-5
allocation of the SGA, 6-2
starting address, 6-13
exclusive mode, 5-6
fast warmstart, 28-4
forcing, 5-6
prohibited by dispatcher processes, 7-24
recovery during, 28-4
restricted mode, 5-5
shared mode, 5-6
steps, 5-5
statement triggers, 18-7
described, 18-8
when fired, 18-14
See also triggers
statement-level read consistency, 23-5
statements
See SQL statements
static ownership, 31-23
statistics
ANALYZE command, 20-41
checkpoint, 7-11
optimizer use of, 20-6, 20-7
partitioned tables and indexes, 9-12
queuing, 16-8
storage
datafiles, 3-11
for parallel INSERT, 21-8
fragmentation in parallel DDL, 22-28
logical structures, 3-3, 8-2
nested tables, 12-5
object tables, 12-4
of hash clusters, 8-37
of index clusters, 8-34
of index partitions, 9-29
of indexes, 8-19
of nulls, 8-7
of table partitions, 9-21
of views, 8-12
REFs, 12-5
restricting for users, 25-8
revoking tablespaces and, 25-9
tablespace quotas and, 25-8
triggers, 18-2, 18-17
user quotas on, 1-32
STORAGE clause
parallel query, 22-27
using, 2-11
storage parameters
NEXT, 21-8
OPTIMAL (in rollback segments), 2-22
PCTINCREASE, 21-8
setting, 2-11
store-and-forward replication, 31-19
stored functions, 1-44, 17-2, 17-6
stored procedures, 1-44, 14-15, 17-2, 17-6
calling, 14-18
contrasted with anonymous blocks, 17-8
triggers contrasted with, 18-2
variables and constants, 14-17
See also procedures
Structured Query Language (SQL), 1-48, 14-2
See also SQL

structures
locking, 23-26
logical, 1-6, 1-8
tablespaces, 3-3
physical, 1-5, 1-11
datafiles, 3-11
schema objects, 8-2
subqueries, 14-12
CHECK constraints prohibit, 24-16
converting to joins, 20-22
in DML statements
serializable isolation, 23-13
in remote updates, 30-10
NOT IN, 20-74
optimizing IN subquery, 20-25
See also queries
subquery snapshot, 31-7
SunSoft
SunNet Manager, 30-19
survivability, 28-24
symmetric replication, 31-3, 31-12
overview, 31-11
uses for, 31-12
synchronous data propagation, 31-27
synonyms, 19-7
constraints indirectly affect, 24-5
described, 8-15
for data dictionary views, 4-4
inherit privileges from object, 26-3
overview of, 1-44
private, 8-16
public, 8-16
uses of, 8-16
SYS username
audit records not generated by, 27-5
data dictionary tables owned by, 4-3
security domain of, 25-3
SYS.AUD$ view
purging, 4-5
SYSDBA privilege, 5-3
SYSOPER privilege, 5-3
system change numbers (SCN)
committed transactions, 15-5
defined, 15-5
read consistency and, 23-5
redo logs, 7-10
when determined, 23-5
system control statements, 1-49, 14-5
system global area (SGA), 6-2
allocating, 5-5
contents of, 6-3
data dictionary cache, 4-4
database buffer cache, 6-3
diagram, 5-2
fixed, 6-3
limiting use of in multithreaded server, 25-12
overview of, 1-15, 6-2
redo log buffer, 6-6, 15-5
rollback segments and, 15-5
shared and writable, 6-3
shared pool, 6-6
size of, 6-11
variable parameters, 5-4
when allocated, 6-2
system monitor process (SMON), 7-11
defined, 1-18, 7-11
instance recovery, 28-4
parallel DML instance recovery, 22-35
parallel DML system recovery, 22-35
Parallel Server and, 7-11, 22-35
rolling back transactions, 28-10
temporary segment cleanup, 7-11
system privileges, 26-2
ADMIN OPTION, 12-11, 26-3
described, 26-2
granting and revoking, 26-3
user-defined types, 12-10
See also privileges
SYSTEM rollback segment, 2-23
SYSTEM tablespace, 3-4
data dictionary stored in, 4-2, 4-5
media failure, 28-6
online requirement of, 3-7
procedures stored in, 17-16
SYSTEM username
security domain of, 25-3

T

table directories, 2-4
tables
  affect dependent views, 19-5
  auditing, 9-42, 27-8
  base, 1-43
    data dictionary use of, 4-2
    relationship to views, 8-11
  clustered, 8-32
  contain integrity constraints, 1-54
  contained in tablespaces, 8-5
  controlling space allocation for, 8-4, 21-8
  DUAL, 4-7
  dynamic partitioning, 22-6
  enable or disable constraints, 24-21
  full table scan and buffer cache, 6-4
  hash, 8-41
  historical, 22-32
  how data is stored in, 8-4
  indexes and, 8-17
  index-organized tables, 8-28
  integrity constraints, 24-2, 24-5
  locks on, 9-30, 23-20, 23-22, 23-23
  maximum number of columns in, 8-10
  nested tables, 8-9, 11-10
    indexes, 12-9
  no-logging mode, 21-7
  novalidate constraints, 24-21
  object tables, 11-3, 11-7
    constraints, 12-8
    indexes, 12-9
    triggers, 12-9
    virtual, 13-2
  overview of, 1-42, 8-3
  parallel creation, 22-26
  parallel DDL storage, 22-27
  parallel table scans, 22-4
  partition-extended table names, 9-42
  partitions, 9-2, 9-21
  presented in views, 8-10
  privileges for partitions, 9-41
  privileges on, 26-4
  queue tables, 16-4, 16-8
  refreshing in data warehouse, 22-31
  replicating, 1-26, 31-2, 31-3
  specifying tablespaces for, 8-5
  STORAGE clause with parallel query, 22-27
  summary or rollup, 22-26
  table aliases, 12-2, 12-3
  table names
    qualifying column names, 12-2, 12-3
  triggers used in, 18-2
  virtual or viewed, 1-43
tablespaces, 3-3
  contrasted with schemas, 8-2
  default for object creation, 1-32, 25-8
  described, 3-3
  how specified for tables, 8-5
  locks on, 23-29
  no-logging mode, 21-7
  offline, 1-9, 3-7, 3-12
    and index data, 3-9
    cannot be read-only, 3-10
    remain offline on remount, 3-8
  online, 1-9, 3-7, 3-12
  overview of, 1-9, 3-3
  quotas on, 1-31, 1-32, 25-8
    limited and unlimited, 25-8
    no default, 25-8
  read-only, 3-9
    dropping objects from, 3-10
  relationship to datafiles, 3-2
  revoking access from users, 25-9
  size of, 3-6
  temporary, 1-32, 3-11
    default for user, 25-8
  used for temporary segments, 2-14, 2-16
tasks, 7-2
temporary segments, 2-14, 2-16
  allocating, 2-16
  deallocating extents from, 2-14
  dropping, 2-14
  ignore quotas, 25-9
  operations that require, 2-16
  parallel DDL, 22-28
  parallel INSERT, 21-8
  tablespace containing, 2-14, 2-16
  when not in redo log, 2-17
temporary tablespaces, 3-11

threads
multithreaded server, 7-13, 7-20
three-valued logic (true, false, unknown)
produced by nulls, 8-7
throughput, 20-7
cost-based approach, 20-40
timestamp checking, 19-9
TO_DATE function, 10-7
partition pruning, 9-12, 9-16
trace files, 7-14
DBW*n*, 28-6
LGWR trace file, 7-10
transaction control statements, 1-49, 14-5
transaction set consistency, 23-9, 23-10
transaction tables, 2-18
reset at recovery, 7-12
transactions, 1-49, 15-1
advanced queuing, 16-3
assigning system change numbers, 15-5
assigning to rollback segments, 2-18
asynchronous processing, 16-2
committing, 1-51, 7-10, 15-3, 15-5
group commits, 7-10
use of rollback segments, 2-19
concurrency and, 23-15
controlling transactions, 14-14
deadlocks and, 15-4, 23-16
defining and controlling, 14-14
described, 15-2
discrete transactions, 14-14, 15-8
distributed, 1-22
deadlocks and, 23-17
parallel DML restrictions, 22-40
resolving automatically, 7-12
two-phase commit, 1-25, 15-7, 30-12
distribution among rollback segments of, 2-19
end of, 15-4
consistent data, 14-14
in-doubt
limit rollback segment access, 2-27
resolving automatically, 1-25, 5-7, 15-8
resolving manually, 1-26
rollback segments and, 2-21
use partly available segments, 2-27
manual locking of, 23-30
overview of, 1-49
read consistency of, 1-21, 23-6
read-only, 1-22, 23-6
not assigned to rollback segments, 2-18
redo log files written before commit, 7-10
rollback segments and, 2-18
rolling back, 1-51, 15-6
and offline tablespaces, 2-27
partially, 15-6
use of rollback segments, 2-18
savepoints in, 1-51, 15-7
serializable, 23-6
space used in data blocks for, 2-5
start of, 15-4
statement level rollback and, 15-4
system change numbers, 7-10
terminating the application and, 15-5
transaction control statements, 14-5
triggers and, 18-16
two-phase commit in parallel DML, 22-34
writing to rollback segments, 2-19
TRANSACTIONS parameter, 2-24
TRANSACTIONS_PER_ROLLBACK_SEGMENT parameter, 2-24
triggers, 1-55, 18-1, 19-7
action, 18-7
timing of, 18-8
AFTER triggers, 18-9
as program units, 1-53
auditing, 27-8
BEFORE triggers, 18-8
cascading, 18-3
constraints apply to, 18-14
constraints contrasted with, 18-5
data access and, 18-16
dependency management of, 18-18, 19-6
enabled triggers, 18-14
enabled or disabled, 18-14
enforcing data integrity with, 24-4
events, 18-6
examples of, 18-10, 18-12, 18-16
firing (executing), 18-2, 18-17
privileges required, 18-17
steps involved, 18-14
timing of, 18-14

triggers *(continued)*
INSTEAD OF triggers, 18-11
object views and, 13-5
INVALID status, 19-3, 19-6
maintain data integrity, 1-55
Oracle Forms triggers vs., 18-4
overview of, 1-55, 18-2
parts of, 18-5
privileges for executing, 26-8
procedures contrasted with, 18-2
prohibited in views, 8-11
restrictions, 18-7, 22-39
direct-load INSERT, 21-10
parallel DML, 22-38
roles disabled in, 26-14
row, 18-8
schema object dependencies, 18-14, 18-18
sequence for firing multiple, 18-14
shared SQL areas and, 6-9
statement, 18-8
storage of, 18-17
types of, 18-7
UNKNOWN does not fire, 18-7
user-defined types, 12-9
uses of, 18-3
TRUNCATE command, 14-4
Trusted Oracle
described, 1-34
mandatory access control, 1-34
MLSLABEL datatype, 10-16
mounting multiple databases in, 5-3
two-phase commit
described, 1-25, 30-12
manual override of, 1-26
parallel DML, 22-34
transaction management, 15-7
triggers, 18-14
two-task mode, 7-16, 7-18
described, 7-18
listener process and, 7-14
network communication and, 7-19
program interface in, 7-19
types
See datatypes, object types

U

undo, 1-11
See also rollback
UNION ALL operator
examples, 20-20, 20-22, 20-36
optimizing view queries, 20-25
transforming OR into, 20-19
UNION ALL views, 9-10
UNION operator
compound queries, 20-14
examples, 20-27, 20-37
optimizing view queries, 20-25
unique indexes, 8-17
UNIQUE key constraints, 24-8
composite keys, 24-8, 24-10
constraint checking, 24-19
indexes used to enforce, 24-9
maximum number of columns, 24-9
NOT NULL constraints and, 24-10
nulls and, 24-10
size limit of, 24-9
unique keys, 1-54, 1-55, 24-8
composite, 24-8, 24-10
optimization, 20-23
searches, 20-48
uniqueness conflict, 31-23
updatable snapshot, 31-14
UPDATE command, 14-4
foreign key references and, 24-15
freeing space in data blocks, 2-9
parallel UPDATE, 22-18
triggers and, 18-2, 18-6
BEFORE triggers, 18-8
INSTEAD OF triggers, 18-11
update no action constraint, 24-15
UPDATE privilege for object tables, 12-12, 12-13
updates
conflict in replicated data, 31-23
distributed, 30-11
location transparency and, 30-14
object views, 13-4
remote, 30-10
updatability of object views, 13-4

updates *(continued)*
updatability of views, 8-13, 18-11, 18-12
updatable join views, 8-13
update intensive environments, 23-8
USE_INDIRECT_DATA_BUFFERS parameter, 6-13
user locks, 23-40
user processes
allocate PGAs, 6-13
connections and, 7-4
dedicated server processes and, 7-18
sessions and, 7-5
shared server processes and, 7-23
user program interface (UPI), 7-27
USER pseudocolumn, 26-7
USER_ views, 4-6
USER_TAB_COLUMNS view, 20-60
USER_TABLES view, 20-60
USER_UPDATABLE_COLUMNS view, 8-14
user-defined datatypes, 11-1, 11-3, 12-1
collections, 11-9
nested tables, 11-10
variable arrays (VARRAYs), 11-10
Export and Import, 12-15
incomplete types, 12-13
object types, 11-2, 11-4
use of table aliases, 12-2
object-relational model, 1-41
privileges, 12-10
storage, 12-4
users, 25-2
access rights, 25-2
associated with schemas, 8-2
auditing, 27-12
authentication of, 25-3
coordinating concurrent actions of, 1-20
dedicated servers and, 7-18
default tablespaces of, 25-8
distributed databases, 30-16
licensing by number of, 25-15
licensing of, 25-14
listed in data dictionary, 4-2
multiuser environments, 1-2, 7-3
password encryption, 25-4
privileges of, 1-30
processes of, 1-16, 7-4
profiles of, 1-32, 25-13
PUBLIC user group, 25-9, 26-14
resource limits of, 25-10
restrictions on resource use of, 1-31
roles and, 26-10
for types of users, 26-12
schemas of, 1-28, 25-2
security domains of, 1-29, 25-2, 26-14
single-user Oracle, 7-2
tablespace quotas of, 1-32, 25-8
tablespaces of, 1-32
temporary tablespaces of, 1-32, 2-16, 25-8
usernames, 1-29, 25-2
sessions and connections, 7-5

V

V_$ and V$ views, 4-7
V$LICENSE, 25-15
VARCHAR datatype, 10-3
VARCHAR2 datatype, 10-3
non-padded comparison semantics, 10-3
similarity to RAW datatype, 10-11
variables
bind variables
optimization, 20-60
user-defined types, 11-12
embedded SQL, 14-6
in stored procedures, 14-17
object variables, 13-4
VARRAYs, 11-10
very large database (VLDB), 9-4
parallel SQL, 22-2
partitions, 9-4
views, 1-43, 8-10
altering base tables and, 19-5
auditing, 27-8
base tables, 1-43
complex view merging, 20-25
constraints and triggers prohibited in, 8-11
constraints indirectly affect, 24-5
containing expressions, 18-12

views *(continued)*
data dictionary
updatable columns, 8-13
user-accessible views, 4-3
definition expanded, 19-5
dependency status of, 19-5
histograms, 20-10
how stored, 8-11
indexes and, 8-12
inherently modifiable, 18-12
INVALID status, 19-3
maximum number of columns in, 8-10
modifiable, 18-12
modifying, 18-11
NLS parameters in, 8-12
non-null values for nulls, 20-72
object views, 8-14, 13-1
row objects, 11-8
updatability, 13-4
optimization, 20-24
overview of, 1-43, 8-10
partition statistics, 9-12
partition views, 9-10
prerequisites for compilation of, 19-5
privileges for, 26-6
pseudocolumns, 18-12
schema object dependencies, 8-13, 19-4, 19-7
security applications of, 26-6
select-project-join views, 20-24
SQL functions in, 8-12
updatability, 8-13, 13-4, 18-12
uses of, 8-11
virtual memory, 6-16
virtual tables, 1-43
VLDB
parallel SQL, 22-2
partitions, 9-4

W

waits for blocking transaction, 23-10
warehouse
refreshing table data, 22-31
See also data warehousing
whole database backups, 1-39, 28-21
WITH OBJECT OID clause, 13-3, 13-4
workload skewing, 22-16
wrapper
procedural replication, 31-26
write-ahead, 7-10
writers block readers, 23-10

Y

year 2000, 10-9