Oracle8*i*

Designing and Tuning for Performance

Release 2 (8.1.6)

December 1999

Part No. A76992-01

ORACLE

Designing and Tuning for Performance, Release 2 (8.1.6)

Part No. A76992-01

Copyright © 1996, 1999, Oracle Corporation. All rights reserved.

Primary Author: Michele Cyran.

Contributing Authors: Mark Bauer, Ruth Baylis, Lance Ashdown, Joyce Fee, Jackie Gosselin, Shelley Higgins, Diana Lorentz, Rita Moran, Randy Urbano, Nitin Vengurlekar, and Sandy Venning.

Contributors: T. Akiba, Ahmed Alomari, D. Austin, A. Brumm, D. Colello, B. Dageville, D. Daniels, Dinesh Das, S. DeMel, Harv Heneman, S. Gossett, T. Guay, G. Hallmark, M. Hartstein, S. Heisey, A. Ho, Andrew Holdsworth, Hakan Jakobssen, S. Jang, R. Jenkins, J. Klokkers, A. Kolk, Tirthankar Lahiri, J. Loaiza, G. Lumpkin, R. Manalac, S. Maring, Alan Maxwell, K. Morse, Ari Mozes, K. Ono, Cetin Ozbutun, Peter Povinec, M. Rhodes, R. Roccaforte, H. Sankar, R. Shah, Ekrem Soylemez, Juan Tellez, Bob Thome, L. To, A. Tsukerman, Steve Vivian, S. Wadhwa, Steve Wertheimer, Graham Wood, M. Zait, Zia Ziauddin.

Graphic Designer: Valarie Moore

The Programs (which include both the software and documentation) contain proprietary information of Oracle Corporation; they are provided under a license agreement containing restrictions on use and disclosure and are also protected by copyright, patent, and other intellectual and industrial property laws. Reverse engineering, disassembly, or decompilation of the Programs 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. Except as may be expressly permitted in your license agreement for these Programs, no part of these Programs may be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose, without the express written permission of Oracle Corporation. If the Programs are delivered to the U.S. Government or anyone licensing or using the programs on behalf of the U.S. Government, the following notice is applicable:

Restricted Rights Notice Programs delivered subject to the DOD FAR Supplement are "commercial computer software" and use, duplication, and disclosure of the Programs, including documentation, 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 52.227-19, Commercial Computer Software - Restricted Rights (June, 1987). Oracle Corporation, 500 Oracle Parkway, Redwood City, CA 94065.

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

Oracle is a registered trademark, and JServer, LogMiner, Net8, Oracle Advanced Queuing, Oracle Call Interface, Oracle COM Cartridge, Oracle Data Migration Assistant, Oracle Database Assistant, Oracle Database Configuration Assistant, Oracle DBA Management Pack, Oracle Designer, Oracle Developer, Oracle Enterprise Manager, Oracle Enterprise Manager Performance Pack, Oracle Expert, Oracle *i*FS, Oracle *inter*Media, Oracle Lite, Oracle Parallel Server, Oracle Spatial, Oracle Virtual Private Database, Oracle Visual Information Retrieval (VIR), Oracle Web Application Server, Oracle WebDB, Oracle7, Oracle8, Oracle8 Enterprise Edition, Oracle8*i*, Oracle8*i* Lite, PL/SQL, Pro*C, Pro*C/C++, Pro*COBOL, SQL, SQL*Loader, SQL*Net, SQL*Plus, and Wallet Manager are trademarks or registered trademarks of Oracle Corporation. All other company or product names mentioned are used for identification purposes only and may be trademarks of their respective owners.

Contents

Send Us Your Comments xvii

Preface xix

- **Intended Audience** xx
- **How This Book is Organized** xx
- **What's New** xxii
- **Related Documents** xxii
- **Conventions** xxiii
  - Text xxiii
  - Syntax Diagrams and Notation xxiii
  - Code Examples xxv

Part I Introduction to Tuning

1 Understanding Oracle Performance Tuning

- **What Is Performance Tuning?** 1-2
  - Trade-offs Between Response Time and Throughput 1-2
  - Critical Resources 1-4
  - Effects of Excessive Demand 1-5
  - Adjustments to Relieve Problems 1-6
- **Who Tunes?** 1-7
- **Setting Performance Targets** 1-9
- **Setting User Expectations** 1-9
- **Evaluating Performance** 1-10

2 Performance Tuning Methods

When Is Tuning Most Effective? 2-2
Proactive Tuning While Designing and Developing Systems 2-2
Reactive Tuning to Improve Production Systems 2-3
Prioritized Tuning Steps 2-5
Step 1: Tune the Business Rules 2-7
Step 2: Tune the Data Design 2-8
Step 3: Tune the Application Design 2-9
Step 4: Tune the Logical Structure of the Database 2-9
Step 5: Tune Database Operations 2-9
Step 6: Tune the Access Paths 2-11
Step 7: Tune Memory Allocation 2-11
Step 8: Tune I/O and Physical Structure 2-12
Step 9: Tune Resource Contention 2-13
Step 10: Tune the Underlying Platform(s) 2-13
Applying the Tuning Method 2-14
Set Clear Goals for Tuning 2-14
Create Minimum Repeatable Tests 2-14
Test Hypotheses 2-15
Keep Records and Automate Testing 2-15
Avoid Common Errors 2-15
Stop Tuning When Objectives Are Met 2-16
Demonstrate Meeting the Objectives 2-16

Part II Application Design Tuning for Designers and Developers

3 Application and System Performance Characteristics

Types of Applications 3-2
Online Transaction Processing (OLTP) 3-2
Decision Support Systems 3-4
Multipurpose Applications 3-6
Registering Applications 3-7
Oracle Configurations 3-7
Distributed Systems 3-8

Multi-Tier Systems 3-9
Oracle Parallel Server 3-10
Client/Server Configurations 3-11

4 The Optimizer

SQL Processing Architecture 4-2
Parser 4-3
Optimizer 4-3
Row Source Generator 4-3
SQL Execution 4-3
EXPLAIN PLAN 4-3
What Is The Optimizer? 4-4
Execution Plan 4-5
Choosing an Optimizer Approach and Goal 4-8
OPTIMIZER_MODE Initialization Parameter 4-10
Statistics in the Data Dictionary 4-10
OPTIMIZER_GOAL Parameter of the ALTER SESSION Statement 4-11
Changing the Goal with Hints 4-11
Cost-Based Optimizer (CBO) 4-12
Architecture of the CBO 4-13
Features that Require the CBO 4-19
Using the CBO 4-20
Access Paths for the CBO 4-20
How the CBO Chooses an Access Path 4-25
CBO Parameters 4-28
Parameters Affecting CBO Plans 4-29
Parameters Affecting How the Optimizer Uses Indexes 4-30
Setting Initialization Parameters 4-31
Extensible Optimizer 4-32
User-Defined Statistics 4-33
User-Defined Selectivity 4-33
User-Defined Costs 4-33
Rule-Based Optimizer (RBO) 4-34
Access Paths for the RBO 4-34
Overview of Optimizer Operations 4-47

Types of SQL Statements 4-47
Optimizer Operations 4-48
Optimizing Joins 4-49
Optimizing Join Statements 4-49
Join Operations 4-50
How the Optimizer Chooses the Join Method 4-56
Forcing the Join Order 4-57
Choosing Execution Plans for Join Statements 4-58
Optimizing Anti-Joins and Semi-Joins 4-61
Optimizing Star Queries 4-62
Optimizing Statements that Use Common Subexpressions 4-63
Evaluation of Expressions and Conditions 4-65
Constants 4-65
LIKE Operator 4-66
IN Operator 4-66
ANY or SOME Operator 4-67
ALL Operator 4-67
BETWEEN Operator 4-68
NOT Operator 4-68
Transitivity 4-69
DETERMINISTIC Functions 4-70
Transforming and Optimizing Statements 4-71
Transforming ORs into Compound Queries 4-71
Transforming Complex Statements into Join Statements 4-74
Optimizing Statements That Access Views 4-76
Optimizing Compound Queries 4-91
Optimizing Distributed Statements 4-94

5 Using EXPLAIN PLAN

Understanding EXPLAIN PLAN 5-2
Creating the Output Table 5-3
Displaying PLAN_TABLE Output 5-4
Output Table Columns 5-4
Bitmap Indexes and EXPLAIN PLAN 5-13
EXPLAIN PLAN and Partitioned Objects 5-14

Displaying Range and Hash Partitioning with EXPLAIN PLAN 5-14
Pruning Information with Composite Partitioned Objects 5-16
Partial Partition-wise Joins 5-19
Full Partition-wise Joins 5-20
INLIST ITERATOR and EXPLAIN PLAN 5-21
Domain Indexes and EXPLAIN PLAN 5-22
EXPLAIN PLAN Restrictions 5-23

6 Using SQL Trace and TKPROF

Understanding SQL Trace and TKPROF 6-2
Understanding the SQL Trace Facility 6-2
Understanding TKPROF 6-3
Using the SQL Trace Facility and TKPROF 6-3
Step 1: Setting Initialization Parameters for Trace File Management 6-4
Step 2: Enabling the SQL Trace Facility 6-5
Step 3: Formatting Trace Files with TKPROF 6-6
Step 4: Interpreting TKPROF Output 6-11
Step 5: Storing SQL Trace Facility Statistics 6-16
Avoiding Pitfalls in TKPROF Interpretation 6-19
The Argument Trap 6-19
The Read Consistency Trap 6-19
The Schema Trap 6-20
The Time Trap 6-21
The Trigger Trap 6-22
TKPROF Output Example 6-22
Header 6-23
Body 6-23
Summary 6-29

7 Using Optimizer Hints

Understanding Hints 7-2
Specifying Hints 7-2
Using Hints 7-6
Hints for Optimization Approaches and Goals 7-6
Hints for Access Methods 7-9

Hints for Join Orders 7-18
Hints for Join Operations 7-19
Hints for Parallel Execution 7-24
Additional Hints 7-30
Using Hints with Views 7-36

8 Gathering Statistics

Understanding Statistics 8-2
Generating Statistics 8-3
Using the ANALYZE Statement 8-4
Using the DBMS_STATS Package 8-5
Statistics Data 8-10
Missing Statistics 8-11
Using Statistics 8-12
Managing Statistics 8-12
Verifying Table Statistics 8-13
Verifying Index Statistics 8-14
Verifying Column Statistics 8-15
Using Histograms 8-17
When to Use Histograms 8-18
Creating Histograms 8-18
Types of Histograms 8-19
Viewing Histograms 8-21
Verifying Histogram Statistics 8-21

9 Optimizing SQL Statements

Approaches to SQL Statement Tuning 9-2
Restructuring the Indexes 9-2
Restructuring the Statement 9-2
Modifying or Disabling Triggers 9-12
Restructuring the Data 9-12
Keeping Statistics Current and Using Plan Stability to Preserve Execution Plans 9-13
Tuning Goals 9-13
Tuning a Serial SQL Statement 9-14
Tuning Parallel Execution 9-14

Tuning OLTP Applications 9-16
Best Practices 9-17
Avoiding Rule-Based Optimizer Techniques 9-17
Index Cost 9-17
Analyzing Object Statistics 9-18
Avoiding Complex Expressions 9-21
Avoiding Balloon Tactic for Coding SQL 9-21
Handling Complex Logic in the Application 9-22
SQL Tuning Tips 9-22
Using EXPLAIN PLAN on All Queries 9-24
Predicate Collapsing 9-24
Tuning for the Typical Case 9-25
Disk Reads and Buffer Gets 9-27
Using EXISTS versus IN 9-28
Trouble Shooting 9-29
Tuning Distributed Queries 9-30
Remote and Distributed Queries 9-30
Distributed Query Restrictions 9-40
Transparent Gateways 9-40
Optimizing Performance of Distributed Queries 9-41

10 Using Plan Stability

Using Plan Stability to Preserve Execution Plans 10-2
Hints and Exact Text Matching 10-2
Storing Outlines 10-4
Enabling Plan Stability 10-4
Creating Outlines 10-4
Using Stored Outlines 10-5
Viewing Outline Data 10-6
Using the OUTLN_PKG Package to Manage Stored Outlines 10-7
Moving Outline Tables 10-7
Plan Stability Procedures for the Cost-Based Optimizer 10-8
Using Outlines to Move to the Cost-Based Optimizer 10-8
RDBMS Upgrades and the Cost-Based Optimizer 10-9

Part III Application Design Tools for Designers and DBAs

11 Overview of Diagnostic Tools

Sources of Data for Tuning 11-2
Data Volumes 11-2
Online Data Dictionary 11-3
Operating System Tools 11-3
Dynamic Performance Tables 11-3
Oracle Trace and Oracle Trace Data Viewer 11-3
SQL Trace Facility 11-3
Alert Log 11-4
Application Program Output 11-4
Users 11-4
Initialization Parameter Files 11-4
Program Text 11-4
Design (Analysis) Dictionary 11-5
Comparative Data 11-5
Dynamic Performance Views 11-5
Oracle and SNMP Support 11-5
EXPLAIN PLAN 11-6
SQL Trace and TKPROF 11-6
Supported Scripts 11-7
Application Registration 11-8
Oracle Enterprise Manager, Packs, and Applications 11-8
Introduction to Oracle Enterprise Manager 11-9
Oracle Diagnostics Pack 11-10
Oracle Tuning Pack 11-12
Oracle Parallel Server Management 11-14
Independent Tools 11-14

12 Data Access Methods

Using Indexes 12-2
When to Create Indexes 12-2
Tuning the Logical Structure 12-3

Choosing Columns and Expressions to Index 12-4
Choosing Composite Indexes 12-5
Writing Statements that Use Indexes........ 12-6
Writing Statements that Avoid Using Indexes........ 12-7
Assessing the Value of Indexes 12-7
Using Fast Full Index Scans 12-8
Re-creating Indexes 12-9
Compacting Indexes........ 12-10
Using Nonunique Indexes to Enforce Uniqueness........ 12-10
Using Enabled Novalidated Constraints 12-11
Using Function-based Indexes........ 12-12
Function-based Indexes and Index Organized Tables........ 12-13
Using Bitmap Indexes........ 12-13
When to Use Bitmap Indexes........ 12-14
Creating Bitmap Indexes 12-16
Initialization Parameters for Bitmap Indexing........ 12-19
Using Bitmap Access Plans on Regular B*-tree Indexes........ 12-20
Estimating Bitmap Index Size........ 12-21
Bitmap Index Restrictions 12-24
Using Domain Indexes 12-24
Using Clusters 12-25
Using Hash Clusters........ 12-26
When to Use Hash Clusters 12-26
Creating Hash Clusters........ 12-27

13 Managing Shared SQL and PL/SQL Areas

Comparing SQL Statements and PL/SQL Blocks........ 13-2
Testing for Identical SQL Statements 13-2
Aspects of Standardized SQL Formatting 13-3
Keeping Shared SQL and PL/SQL in the Shared Pool........ 13-3
Reserving Space for Large Allocations........ 13-3
Preventing Objects from Aging Out 13-4

14 Using Oracle Trace

Introduction to Oracle Trace 14-2
Using Oracle Trace Data 14-2
Using Oracle Trace Manager 14-4
Managing Collections 14-4
Collecting Event Data 14-5
Accessing Collected Data 14-5
Using Oracle Trace Data Viewer 14-6
Oracle Trace Predefined Data Views 14-6
Viewing Oracle Trace Data 14-13
SQL Statement Property Page 14-15
Details Property Page 14-15
Example of Details Property Page 14-15
Getting More Information on a Selected Query 14-17
Manually Collecting Oracle Trace Data 14-20
Using the Oracle Trace Command-Line Interface 14-20
Using Initialization Parameters to Control Oracle Trace 14-22
Using Stored Procedures to Control Oracle Trace 14-25
Oracle Trace Collection Results 14-27
Formatting Oracle Trace Data to Oracle Tables 14-27
Oracle Trace Statistics Reporting Utility 14-28

15 Dynamic Performance Views

Instance-Level Views for Tuning 15-2
Session-Level or Transient Views for Tuning 15-3
Current Statistic Values and Rates of Change 15-3
Finding the Current Value of a Statistic 15-4
Finding the Rate of Change of a Statistic 15-4

16 Diagnosing System Performance Problems

Tuning Factors for Well Designed Existing Systems 16-2
Insufficient CPU 16-5
Insufficient Memory 16-5
I/O Constraints 16-6

Network Constraints 16-6
Software Constraints 16-7

17 Transaction Modes

Using Discrete Transactions 17-2
Deciding When to Use Discrete Transactions 17-2
How Discrete Transactions Work 17-3
Errors During Discrete Transactions 17-3
Using Discrete Transactions 17-3
Example 17-4
Using Serializable Transactions 17-6

Part IV Optimizing Instance Performance

18 Tuning CPU Resources

Understanding CPU Problems 18-2
Detecting and Solving CPU Problems 18-4
System CPU Utilization 18-4
Oracle CPU Utilization 18-6
Solving CPU Problems by Changing System Architectures 18-13
Single Tier to Two-Tier 18-14
Multi-Tier: Using Smaller Client Machines 18-15
Two-Tier to Three-Tier 18-15
Three-Tier 18-17
Oracle Parallel Server 18-17

19 Tuning Memory Allocation

Understanding Memory Allocation Issues 19-2
Detecting Memory Allocation Problems 19-3
Solving Memory Allocation Problems 19-3
Tuning Operating System Memory Requirements 19-4
Tuning the Redo Log Buffer 19-6
Tuning Private SQL and PL/SQL Areas 19-8
Tuning the Shared Pool 19-11

Tuning the Buffer Cache 19-27
Tuning Multiple Buffer Pools 19-32
Tuning Sort Areas 19-41
Reallocating Memory 19-42
Reducing Total Memory Usage 19-42

20 Tuning I/O

Understanding I/O Problems 20-2
Tuning I/O: Top Down and Bottom Up 20-2
Analyzing I/O Requirements 20-3
Planning File Storage 20-5
Choosing Data Block Size 20-10
Evaluating Device Bandwidth 20-11
Detecting I/O Problems 20-15
Checking System I/O Utilization 20-15
Checking Oracle I/O Utilization 20-15
Solving I/O Problems 20-18
Reducing Disk Contention by Distributing I/O 20-18
Striping Disks 20-22
Avoiding Dynamic Space Management 20-26
Tuning Sorts 20-35
Tuning Checkpoint Activity 20-39
Tuning LGWR and DBWR I/O 20-41
Tuning Backup and Restore Operations 20-48
Configuring the Large Pool 20-64

21 Tuning Resource Contention

Understanding Contention Issues 21-2
Detecting Contention Problems 21-2
Solving Contention Problems 21-3
Reducing Contention for Rollback Segments 21-3
Reducing Contention for Multi-Threaded Servers 21-5
Reducing Contention for Parallel Execution Servers 21-14
Reducing Contention for Redo Log Buffer Latches 21-16

Reducing Contention for the LRU Latch 21-19
Reducing Free List Contention 21-20

22 Tuning Networks

Understanding Connection Models 22-2
Detecting Network Problems 22-9
Using Dynamic Performance Views 22-9
Understanding Latency and Bandwidth 22-10
Solving Network Problems 22-11
Finding Bottlenecks 22-12
Dissecting Bottlenecks 22-14
Using Array Interfaces 22-16
Adjusting Session Data Unit Buffer Size 22-16
Using TCP.NODELAY 22-17
Using Connection Manager 22-17

23 Tuning the Operating System

Understanding Operating System Performance Issues 23-2
Operating System and Hardware Caches 23-2
Raw Devices 23-2
Process Schedulers 23-3
Operating System Resource Managers 23-3
Detecting Operating System Problems 23-5
Solving Operating System Problems 23-5
Performance on UNIX-Based Systems 23-6
Performance on NT Systems 23-6
Performance on Mainframe Computers 23-6

24 Tuning Instance Recovery Performance

Understanding Instance Recovery 24-2
How Oracle Applies Redo Log Information 24-2
Trade-offs of Minimizing Recovery Duration 24-2
Tuning the Duration of Instance and Crash Recovery 24-3
Using Initialization Parameters to Influence Recovery Time 24-3

Using Redo Log Size to Influence Checkpointing Frequency 24-6
Using SQL Statements to Initiate Checkpoints 24-7
Monitoring Instance Recovery 24-7
Tuning the Phases of Instance Recovery 24-14
Tuning the Rolling Forward Phase 24-15
Tuning the Rolling Back Phase 24-16

Index

Send Us Your Comments

Designing and Tuning for Performance, Release 8.1.6

Part No. A76992-01

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

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

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

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

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

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

Preface

You can enhance Oracle performance by adjusting database applications, the database, and the operating system. Making such adjustments is known as *tuning*. Proper tuning of Oracle provides the best possible database performance for your specific application and hardware configuration.

Oracle8i Designing and Tuning for Performance contains information describing the features and functionality of the Oracle8*i* and the Oracle8*i* Enterprise Edition products. Oracle8*i* and Oracle8*i* Enterprise Edition have the same basic features. However, several advanced features are available only with the Enterprise Edition, and some of these are optional. For example, to use application failover, you must have the Enterprise Edition and the Parallel Server option.

> **See Also:** For information about the differences between Oracle8*i*, Oracle8*i* Enterprise Edition, and Oracle8*i* Personal Edition, see *Getting to Know Oracle8i.*

This preface includes the following sections:

- Intended Audience
- How This Book is Organized
- What's New
- Related Documents
- Conventions

Intended Audience

This manual is an aid for people responsible for the operation, maintenance, and performance of Oracle. To use this book, you could be a database administrator, application designer, or programmer. You should be familiar with Oracle8*i*, the operating system, and application design before reading this manual.

How This Book is Organized

This book has five parts. The book begins by describing tuning and explaining tuning methods. Part Two describes how system designers and programmers plan for performance. Part Three describes design tools for designers and DBAs. Part Four explains how to optimize performance during production. Part Five describes parallel execution tuning and processing. The contents of the five parts of this manual are:

Part One: Introduction to Tuning

Chapter 1, "Understanding Oracle Performance Tuning"	This chapter provides an overview of tuning issues. It defines performance tuning and the roles of people involved in the process.
Chapter 2, "Performance Tuning Methods"	This chapter presents the recommended tuning method, and outlines its steps in order of priority.

Part Two: Application Design Tuning for Designers and Developers

Chapter 3, "Application and System Performance Characteristics"	This chapter describes the various types of application that use Oracle databases and the suggested approaches and features available when designing each.
Chapter 4, "The Optimizer"	This chapter discusses SQL processing, Oracle optimization, and how the Oracle optimizer chooses how to execute SQL statements.
Chapter 5, "Using EXPLAIN PLAN"	This chapter shows how to use the SQL statement `EXPLAIN PLAN`, and format its output.
Chapter 6, "Using SQL Trace and TKPROF"	This chapter describes the use of the SQL trace facility and `TKPROF`, two basic performance diagnostic tools that can help you monitor and tune applications that run against the Oracle Server.
Chapter 7, "Using Optimizer Hints"	This chapter offers recommendations on how to use cost-based optimizer hints to enhance Oracle performance.

Chapter 8, "Gathering Statistics"	This chapter explains why statistics are important for the cost-based optimizer, and how to gather and use statistics.
Chapter 9, "Optimizing SQL Statements"	This chapter describes how Oracle optimizes Structured Query Language (SQL) using the cost-based optimizer (CBO).
Chapter 10, "Using Plan Stability"	This chapter describes how to use plan stability (stored outlines) to preserve performance characteristics.

Part Three: Application Design Tools for Designers and DBAs

Chapter 11, "Overview of Diagnostic Tools"	This chapter introduces the full range of diagnostic tools available for monitoring production systems and determining performance problems.
Chapter 12, "Data Access Methods"	This chapter provides an overview of data access methods that can enhance performance, and warns of situations to avoid.
Chapter 13, "Managing Shared SQL and PL/SQL Areas"	This chapter explains the use of shared SQL to improve performance.
Chapter 14, "Using Oracle Trace"	This chapter provides an overview of Oracle Trace usage and describes the Oracle Trace initialization parameters.
Chapter 15, "Dynamic Performance Views"	This chapter describes views that are of the greatest use for both performance tuning and ad hoc investigation
Chapter 16, "Diagnosing System Performance Problems"	This chapter provides an overview of performance factors in existing systems that have been properly designed.
Chapter 17, "Transaction Modes"	This chapter describes the different methods in which read consistency is performed.

Part Four: Optimizing Oracle Instance Performance

Chapter 18, "Tuning CPU Resources"	This chapter describes how to identify and solve problems with CPU resources.
Chapter 19, "Tuning Memory Allocation"	This chapter explains how to allocate memory to database structures. Proper sizing of these structures can greatly improve database performance.
Chapter 20, "Tuning I/O"	This chapter explains how to avoid I/O bottlenecks that could prevent Oracle from performing at its maximum potential.

Chapter 21, "Tuning Resource Contention"	This chapter explains how to detect and reduce contention that affects performance.
Chapter 22, "Tuning Networks"	This chapter introduces networking issues that affect tuning, and points to the use of array interfaces, out-of-band breaks, and other tuning techniques.
Chapter 23, "Tuning the Operating System"	This chapter explains how to tune the operating system for optimal performance of Oracle.
Chapter 24, "Tuning Instance Recovery Performance"	This chapter explains how to tune recovery performance.

What's New

For release 8.1.6, this book was renamed *Oracle8i Designing and Tuning for Performance* to emphasize the importance of designing applications and writing SQL properly. Although the goal of the book remains the same, many chapters from release 8.1.5 have been restructured. The main changes with 8.1.6 include the following:

- This manual has expanded information on using the cost-based optimizer (CBO), particularly Chapter 4, "The Optimizer" and Chapter 9, "Optimizing SQL Statements". Some of this information was formerly in the *Oracle8i Concepts* manual.
- Part Five from release 8.1.5 (including information on parallel execution and partitioning) is now part of the new *Oracle8i Data Warehousing Guide*.
- Some information from the 8.1.5 release of this manual was duplicated in other manuals in the Oracle documentation set. This includes information on using the multi-threaded server and using PL/SQL packages. You will find cross-references to those books where this information is provided.

Related Documents

Before reading this manual, you should have already read *Oracle8i Concepts*, the *Oracle8i Application Developer's Guide - Fundamentals*, and the *Oracle8i Administrator's Guide*.

For more information about Oracle Enterprise Manager and its optional applications, see *Oracle Enterprise Manager Concepts Guide, Oracle Enterprise Manager Administrator's Guide*, and *Oracle Enterprise Manager Performance Monitoring and Planning Guide*.

For more information about tuning the Oracle Application Server, see the *Oracle Application Server Performance and Tuning Guide*.

Conventions

This section explains the conventions used in this manual including the following:

- Text
- Syntax Diagrams and Notation
- Code Examples

Text

This section explains the conventions used within the text:

UPPERCASE Characters

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

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

Italicized Characters

Italicized words within text are book titles or emphasized words.

Syntax Diagrams and Notation

The syntax diagrams and notation in this manual show the syntax for SQL statements, functions, hints, and other elements. This section tells you how to read syntax diagrams and examples and write SQL statements based on them.

Keywords

Keywords are words that have special meanings in the SQL language. In the syntax diagrams in this manual, keywords appear in uppercase. You must use keywords in your SQL statements exactly as they appear in the syntax diagram, except that they can be either uppercase or lowercase. For example, you must use the `CREATE` keyword to begin your `CREATE TABLE` statements just as it appears in the `CREATE TABLE` syntax diagram.

Parameters

Parameters act as place holders in syntax diagrams. They appear in lowercase. Parameters are usually names of database objects, Oracle datatype names, or expressions. When you see a parameter in a syntax diagram, substitute an object or expression of the appropriate type in your SQL statement. For example, to write a `CREATE TABLE` statement, use the name of the table you want to create, such as `EMP`, in place of the *table* parameter in the syntax diagram. (Note that parameter names appear in italics in the text.)

This list shows parameters that appear in the syntax diagrams in this manual and examples of the values you might substitute for them in your statements:

Parameter	Description	Examples
table	The substitution value must be the name of an object of the type specified by the parameter.	emp
'text'	The substitution value must be a character literal in single quotes.	'Employee Records'
condition	The substitution value must be a condition that evaluates to TRUE or FALSE.	ename > 'A'
date	The substitution value must be a date constant or an expression of DATE datatype.	`TO_DATE` ('01-Jan-1996', DD-MON-YYYY')
expr	The substitution value can be an expression of any datatype.	sal + 1000
integer	The substitution value must be an integer.	72
rowid	The substitution value must be an expression of datatype ROWID.	`AAAAqYAABAAAEPvAAB`
subquery	The substitution value must be a `SELECT` statement contained in another SQL statement.	`SELECT ename` `FROM emp`
statement_name *block_name*	The substitution value must be an identifier for a SQL statement or PL/SQL block.	s1 b1

Code Examples

SQL and SQL*Plus statements appear separated from the text of paragraphs in a monospaced font. For example:

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

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

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

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

Part I

Introduction to Tuning

Part I provides an overview of Oracle Server tuning concepts. The chapters in this part are:

- Chapter 1, "Understanding Oracle Performance Tuning"
- Chapter 2, "Performance Tuning Methods"

1

Understanding Oracle Performance Tuning

The Oracle server is a sophisticated and highly tunable software product. Its flexibility allows you to make small adjustments that affect database performance. By tuning your system, you can tailor its performance to best meet your needs.

Tuning begins in the system planning and design phases and continues throughout the life of your system. Carefully consider performance issues during the planning phase, and it will be easier to tune your system during production.

This chapter contains the following sections:

- What Is Performance Tuning?
- Who Tunes?
- Setting Performance Targets
- Setting User Expectations
- Evaluating Performance

What Is Performance Tuning?

When considering performance, you should understand several fundamental concepts as described in this section:

- Trade-offs Between Response Time and Throughput
- Critical Resources
- Effects of Excessive Demand
- Adjustments to Relieve Problems

Trade-offs Between Response Time and Throughput

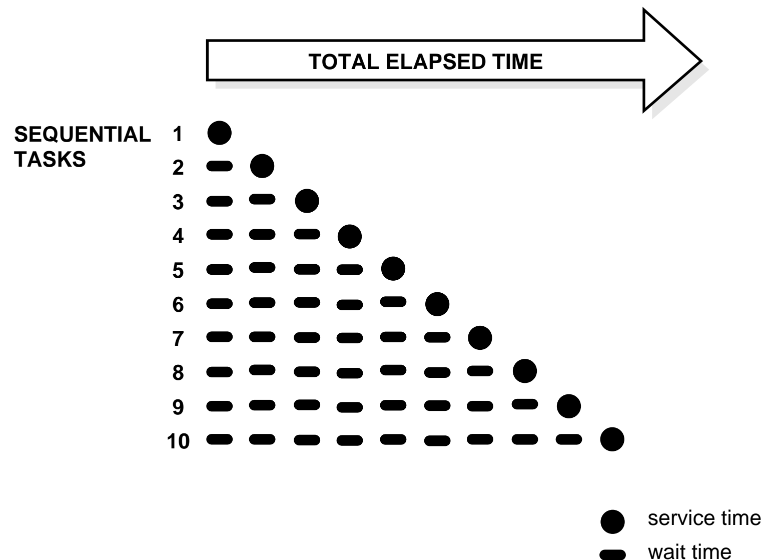
Goals for tuning vary, depending on the needs of the application. Online transaction processing (OLTP) applications define performance in terms of throughput. These applications must process thousands or even millions of very small transactions per day. By contrast, decision support systems (DSS applications) define performance in terms of response time. Demands on the database that are made by users of DSS applications vary dramatically. One moment they may enter a query that fetches only a few records, and the next moment they may enter a massive parallel query that fetches and sorts hundreds of thousands of records from different tables. Throughput becomes more of an issue when an application must support a large number of users running DSS queries.

Response Time

Because response time equals service time plus wait time, you can increase performance two ways: by reducing service time or by reducing wait time.

Figure 1–1 illustrates ten independent tasks competing for a single resource.

Figure 1–1 Sequential Processing of Multiple Independent Tasks



In this example, only task 1 runs without having to wait. Task 2 must wait until task 1 has completed; task 3 must wait until tasks 1 and 2 have completed, and so on. (Although the figure shows the independent tasks as the same size, the size of the tasks vary.)

Note: In parallel processing, if you have multiple resources, then more resources can be assigned to the tasks. Each independent task executes immediately using its own resource: no wait time is involved.

System Throughput

System throughput equals the amount of work accomplished in a given amount of time. Two techniques of increasing throughput exist:

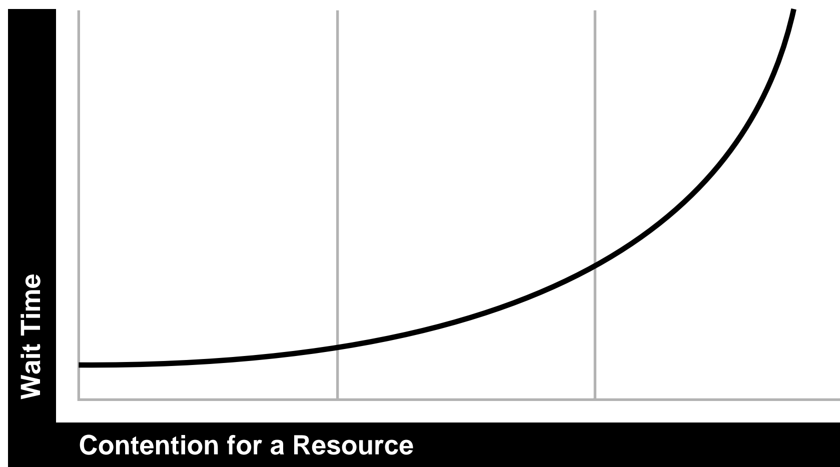
- Get more work done with the same resources (reduce service time).
- Get the work done quicker by reducing overall response time. To do this, look at the wait time. You may be able to duplicate the resource for which all the

users are waiting. For example, if the system is CPU bound, then you can add more CPUs.

Wait Time

The service time for a task may stay the same, but wait time increases as contention increases. If many users are waiting for a service that takes 1 second, then the tenth user must wait 9 seconds for a service that takes 1 second.

Figure 1–2 Wait Time Rising with Increased Contention for a Resource



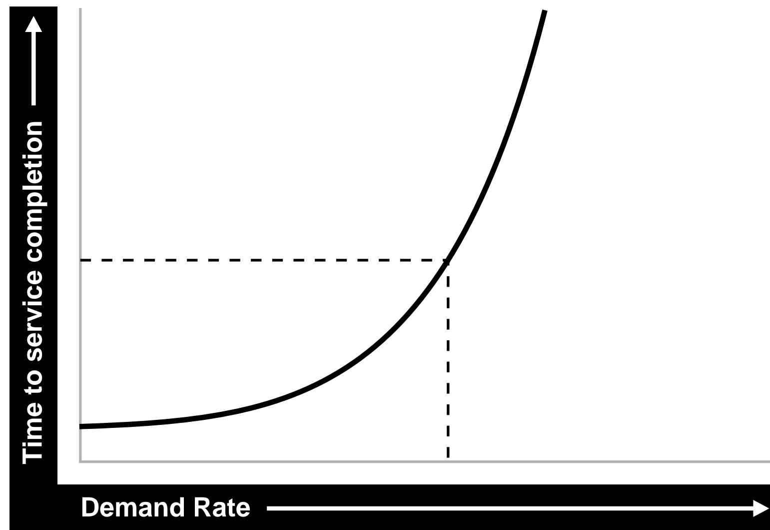
Critical Resources

Resources such as CPUs, memory, I/O capacity, and network bandwidth are key to reducing service time. Added resources make higher throughput possible and facilitate swifter response time. Performance depends on the following:

- How many resources are available?
- How many clients need the resource?
- How long must they wait for the resource?
- How long do they hold the resource?

Figure 1–3 shows that as the number of units requested rises, the time to service completion rises.

Figure 1–3 Time to Service Completion vs. Demand Rate



To manage this situation, you have two options:

- You can limit demand rate to maintain acceptable response times.
- Alternatively, you can add multiple resources: another CPU or disk.

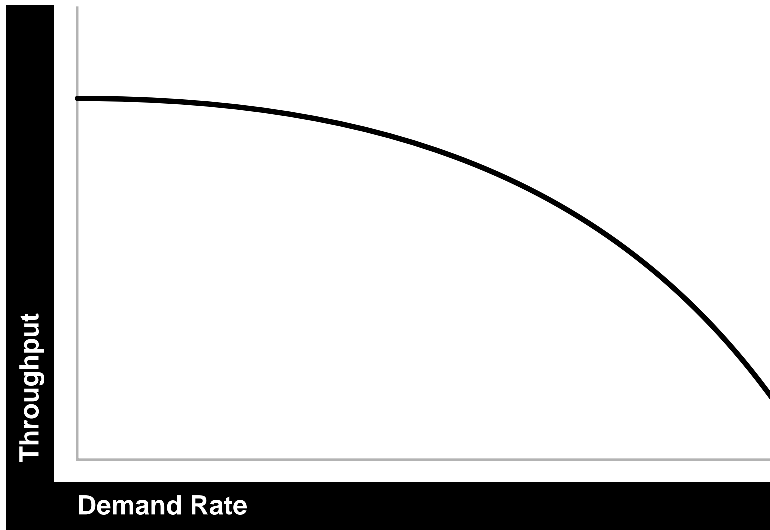
Effects of Excessive Demand

Excessive demand gives rise to the following:

- Greatly increased response time.
- Reduced throughput.

If there is any possibility of demand rate exceeding achievable throughput, then a demand limiter is essential.

Figure 1–4 Increased Response Time/Reduced Throughput



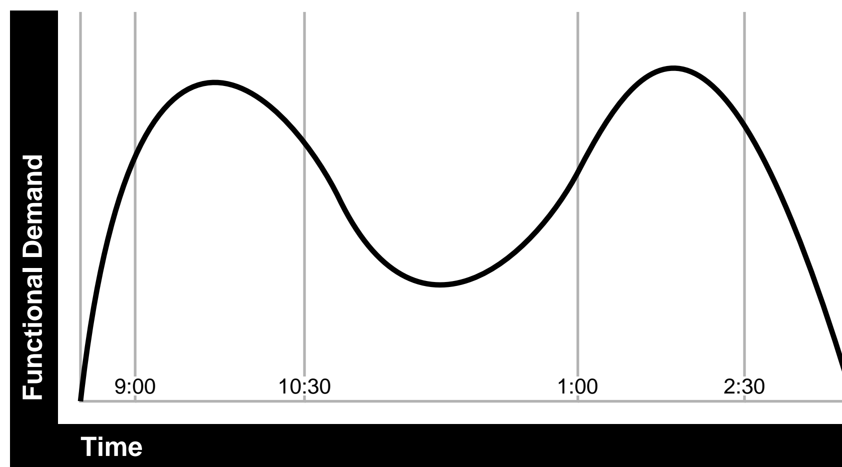
Adjustments to Relieve Problems

You can relieve performance problems by making the following adjustments:

Adjusting unit consumption	You can relieve some problems by using fewer resources per transaction or by reducing service time. Or you can take other approaches, such as reducing the number of I/Os per transaction.
Adjusting functional demand	Other problems can be solved by rescheduling or redistributing the work.
Adjusting capacity	You can also relieve problems by increasing or reallocating resources.

For example, if your system's busiest times are from 9:00AM to 10:30AM and from 1:00PM to 2:30PM, then you can run batch jobs in the background after 2:30PM when there is more capacity. Thus, you can spread the demand more evenly. Alternatively, you can allow for delays at peak times.

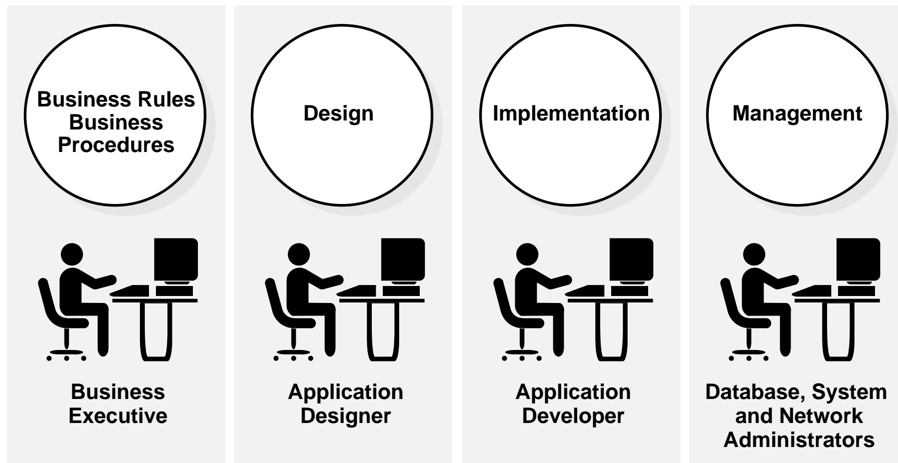
Figure 1–5 Adjusting Capacity and Functional Demand



Who Tunes?

Everyone involved with the system has a role in tuning. When people communicate and document the system's characteristics, tuning becomes significantly easier and faster.

Figure 1–6 Who Tunes the System?



- Business executives must define and then reexamine business rules and procedures to provide a clear and adequate model for application design. They must identify the specific types of rules and procedures that influence the performance of the entire system.
- Application designers must design around potential performance bottlenecks. They must communicate the system design so everyone can understand an application's data flow.
- Application developers must communicate the implementation strategies they select so modules and SQL statements can be quickly and easily identified during statement tuning.
- Database administrators (DBAs) must carefully monitor and document system activity so they can identify and correct unusual system performance. Hardware and software administrators (also know as system administrators and network administrators) must document and communicate the configuration of the system so everyone can design and administer the system effectively.

Decisions made in application development and design have the greatest effect on performance. Once the application is deployed, the database administrator usually has the primary responsibility for tuning.

See Also: Chapter 16, "Diagnosing System Performance Problems" for problem-solving methods that can help identify and solve performance problems.

Setting Performance Targets

Whether you are designing or maintaining a system, you should set specific performance goals so that you know when to tune. You may waste time tuning your system if you alter initialization parameters or SQL statements without a specific goal.

When designing your system, set a goal such as "achieving an order entry response time of less than three seconds for 90% of transactions". If the application does not meet that goal, then identify the bottleneck that prevents this (for example, I/O contention), determine the cause, and take corrective action. During development, test the application to determine whether it meets the designed performance goals before deploying the application.

Tuning is usually a series of trade-offs. Once you have identified bottlenecks, you may need to sacrifice other system resources to achieve the desired results. For example, if I/O is a problem, you may need to purchase more memory or more disks. If a purchase is not possible, then you may need to limit the concurrency of the system to achieve the desired performance. However, with clearly defined performance goals, the decision on what resource to relinquish in exchange for improved performance is simpler because you have identified the most important areas.

Note: At no time should achieving performance goals override your ability to recover data. Performance is important, but ability to recover data is *critical*.

Setting User Expectations

Application developers and database administrators must be careful to set appropriate performance expectations for users. When the system performs a particularly complicated operation, response time may be slower than when it is performing a simple operation. In this case, slower response time is not unreasonable.

If a DBA promises 1-second response time, then consider how this might be interpreted. The DBA might mean that the operation would take 1 second in the

database—and might well be able to achieve this goal. However, users querying over a network might experience a delay of a couple of seconds due to network traffic: they may not receive the response they expect in 1 second.

Evaluating Performance

With clearly defined performance goals, you can readily determine when performance tuning has been successful. Success depends on the functional objectives you have established with the user community, your ability to measure objectively whether the criteria are being met, and your ability to take corrective action to overcome exceptions. The rest of this tuning manual describes the tuning methodology in detail with information about diagnostic tools and the types of corrective actions you can take.

DBAs responsible for solving performance problems must remember all factors that together affect response time. Sometimes what initially seems like the most obvious source of a problem is actually not the problem at all. Users in the preceding example might conclude that there is a problem with the database, whereas the actual problem is with the network. A DBA must monitor the network, disk, CPU, application design, and so on, to identify the actual source of the problem—rather than simply assume that all performance problems stem from the database.

Ongoing performance monitoring enables you to maintain a well-tuned system. You can make useful comparisons by keeping a history of the application's performance over time. Data showing resource consumption for a broad range of load levels helps you conduct objective scalability studies. From such detailed performance history you can begin to predict the resource requirements for future load levels.

See Also: Chapter 11, "Overview of Diagnostic Tools".

2

Performance Tuning Methods

A well-planned methodology is the key to success in performance tuning. Different tuning strategies vary in their effectiveness, and systems with different purposes, such as online transaction processing systems and decision support systems, require different tuning methods.

This chapter contains the following sections:

- When Is Tuning Most Effective?
- Prioritized Tuning Steps
- Applying the Tuning Method

> **See Also:** Oracle Expert automates the process of collecting and analyzing data. It also provides database tuning recommendations, implementation scripts, and performance reports. See Chapter 11, "Overview of Diagnostic Tools" for more information on Oracle Expert.

When Is Tuning Most Effective?

For best results, tune during the design phase, rather than waiting to tune after implementing your system. This is illustrated in the following sections:

- Proactive Tuning While Designing and Developing Systems
- Reactive Tuning to Improve Production Systems

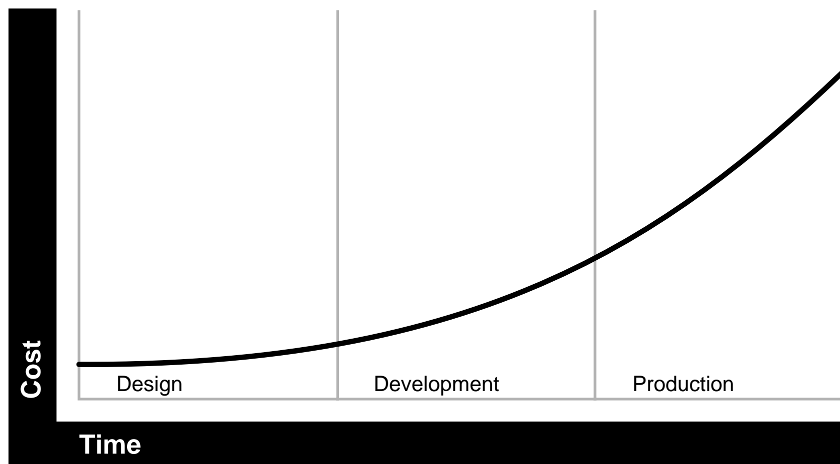
Proactive Tuning While Designing and Developing Systems

By far, the most effective approach to tuning is the proactive approach. Begin by following the steps described in this chapter under "Prioritized Tuning Steps" on page 2-5.

Business executives should work with application designers to establish performance goals and set realistic performance expectations. During design and development, the application designers can then determine which combination of system resources and Oracle features best meet these needs.

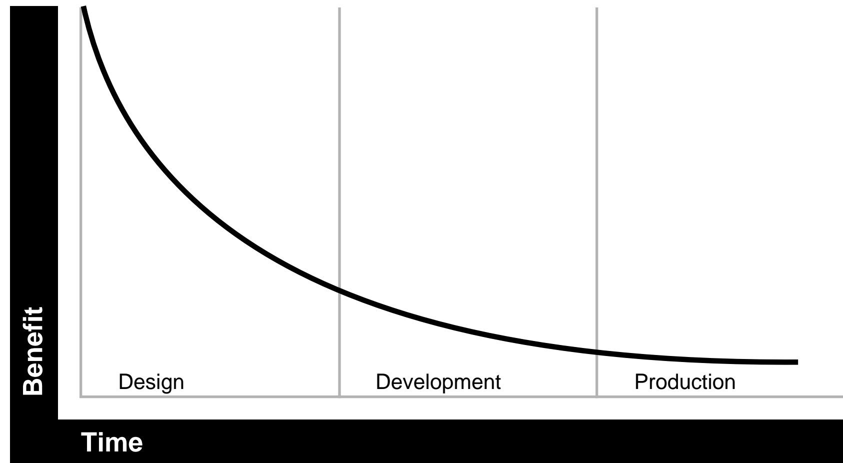
By designing a system to perform well, you can minimize its implementation and on-going administration cost. Figure 2–1 illustrates the relative *cost* of tuning during the life of an application.

Figure 2–1 Cost of Tuning During the Life of an Application



To complement this view, Figure 2–2 shows that the relative *benefit* of tuning an application over the course of its life is inversely proportional to the cost expended.

Figure 2–2 Benefit of Tuning During the Life of an Application



The most effective time to tune is during the design phase: you get the maximum benefit for the lowest cost.

Reactive Tuning to Improve Production Systems

The tuning process does not begin when users complain about poor response time. When response time is this poor, it is usually too late to implement some of the most effective tuning strategies. At that point, if you are unwilling to completely redesign the application, then you may only improve performance marginally by reallocating memory and tuning I/O.

For example: There is a bank that employs one teller and one manager. It has a business rule that the manager must approve withdrawals over $20. You find a long line of customers, and you decide that you need more tellers. You add 10 more tellers, but then you find that the bottleneck moves to the manager's function. However, the bank determines that it is too expensive to hire additional managers. In this example, regardless of how carefully you tune the system using the existing business rule, getting better performance will be very expensive.

Alternatively, a change to the business rule may be necessary to make the system more scalable. If you change the rule so that the manager only needs to approve

withdrawals exceeding $150, then you have created a scalable solution. In this situation, effective tuning could only be done at the highest design level, rather than at the end of the process.

It is possible to reactively tune an existing production system. To take this approach, start at the bottom of the method and work your way up, finding and fixing any bottlenecks. A common goal is to make Oracle run faster on the given platform. You may find, however, that both the Oracle server and the operating system are working well. To get additional performance gains, you may need to tune the application or add resources. Only then can you take full advantage of the many features Oracle provides that can greatly improve performance when properly used in a well-designed system.

Even the performance of well-designed systems can degrade with use. Ongoing tuning is, therefore, an important part of proper system maintenance.

> **See Also:** Part IV, "Optimizing Instance Performance", describes how to tune CPU, memory, I/O, networks, contention, and the operating system.
>
> For background on the Oracle server architecture and features, see *Oracle8i Concepts*.

Prioritized Tuning Steps

The following steps provide a recommended method for tuning an Oracle database. These steps are prioritized in order of diminishing returns: steps with the greatest effect on performance appear first. For optimal results, therefore, resolve tuning issues in the order listed, from the design and development phases through instance tuning.

Step 1: Tune the Business Rules

Step 2: Tune the Data Design

Step 3: Tune the Application Design

Step 4: Tune the Logical Structure of the Database

Step 5: Tune Database Operations

Step 6: Tune the Access Paths

Step 7: Tune Memory Allocation

Step 8: Tune I/O and Physical Structure

Step 9: Tune Resource Contention

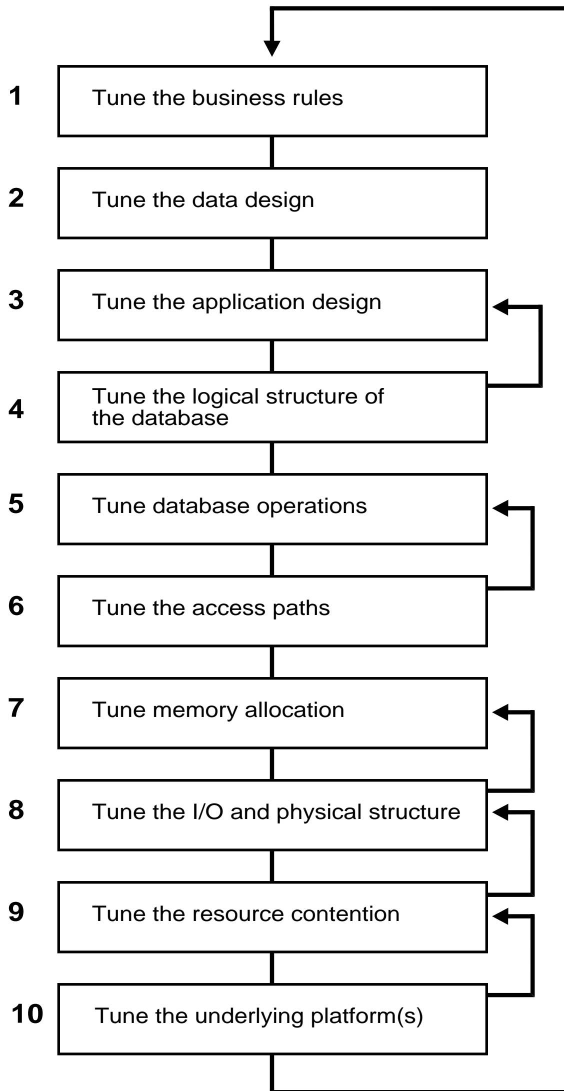
Step 10: Tune the Underlying Platform(s)

After completing these steps, reassess your database performance, and decide whether further tuning is necessary.

Tuning is an iterative process. Performance gains made in later steps may pave the way for further improvements in earlier steps, so additional passes through the tuning process may be useful.

Figure 2–3 illustrates the tuning method:

Figure 2–3 The Tuning Method



Decisions you make in one step may influence subsequent steps. For example, in step 5 you may rewrite some of your SQL statements. These SQL statements may have significant bearing on parsing and caching issues addressed in step 7. Also, disk I/O, which is tuned in step 8, depends on the size of the buffer cache, which is tuned in step 7. Although the figure shows a loop back to step 1, you may need to return from any step to any previous step.

Step 1: Tune the Business Rules

For optimal performance, you may need to adapt business rules. These concern the high-level analysis and design of an entire system. Configuration issues are considered at this level, such as whether to use a multi-threaded server system-wide. In this way, the planners ensure that the performance requirements of the system correspond directly to concrete business needs.

Performance problems encountered by DBAs may actually be caused by problems in design and implementation, or by inappropriate business rules. Designers sometimes provide far greater detail than is needed when they write business functions for an application. They document an implementation, rather than simply the function that must be performed. If business executives effectively distill business functions or requirements from the implementation, then designers have more freedom when selecting an appropriate implementation.

Consider the business function of printing checks. The actual requirement is to pay money to people, not necessarily to print pieces of paper. Whereas it would be very difficult to print a million checks per day, it would be relatively easy to record that many direct deposit payments on a tape that could be sent to the bank for processing.

Business rules should be consistent with realistic expectations for the number of concurrent users, the transaction response time, and the number of records stored online that the system can support. For example, it does not make sense to run a highly interactive application over slow, wide area network lines.

Similarly, a company soliciting users for an Internet service might advertise 10 free hours per month for all new subscribers. If 50,000 users per day signed up for this service, then the demand far exceeds the capacity for a client/server configuration. The company should instead consider using a multi-tier configuration. In addition, the signup process must be simple: it should require only one connection from the user to the database, or connection to multiple databases without dedicated connections, using a multi-threaded server or transaction monitor approach.

Step 2: Tune the Data Design

In the data design phase, you must determine what data is needed by your applications. You must consider what relations are important, and what their attributes are. Finally, you need to structure the information to best meet performance goals.

The database design process generally undergoes a normalization stage when data is analyzed to eliminate data redundancy. With the exception of primary keys, any one data element should be stored only once in your database. After the data is normalized, however, you may need to denormalize it for performance reasons. You might decide that the database should retain frequently used summary values. For example, rather than forcing an application to recalculate the total price of all the lines in a given order each time it is accessed, you might decide to always maintain a number representing the total value for each order in the database. You could set up primary key and foreign key indexes to access this information quickly.

Another data design consideration is avoiding data contention. Consider a database 1 terabyte in size on which one thousand users access only 0.5% of the data. This "hot spot" in the data could cause performance problems.

In a multiple-instance setup, try to localize access to the data down to the partition level, process, and instance levels. That is, localize access to data, such that any process requiring data within a particular set of values is confined to a particular instance. Contention begins when several remote processes simultaneously attempt to access one particular set of data.

In Oracle Parallel Server, look for synchronization points—any point in time, or part of an application that must run sequentially, one process at a time. The requirement of having sequential order numbers, for example, is a synchronization point that results from poor design.

Also consider implementing two Oracle8*i* features that can help avoid contention:

- Consider partitioning your data.
- Consider using local or global indexes.

> **See Also:** For more information on partitioning and indexes, see *Oracle8i Concepts*.

Step 3: Tune the Application Design

Business executives and application designers should translate business goals into an effective system design. Business processes concern a particular application within a system, or a particular part of an application.

An example of intelligent process design is strategically caching data. For example, in a retail application, you can select the tax rate once at the beginning of each day, and cache it within the application. In this way, you avoid retrieving the same information over and over during the day.

At this level, you can also consider the configuration of individual processes. For example, some PC users may access the central system using mobile agents, where other users may be directly connected. Although they are running on the same system, the architecture for each type of user is different. They may also require different mail servers and different versions of the application.

Step 4: Tune the Logical Structure of the Database

After the application and the system have been designed, you can plan the logical structure of the database. This primarily concerns fine-tuning the index design to ensure that the data is neither over- nor under-indexed. In the data design stage (Step 2), you determine the primary and foreign key indexes. In the logical structure design stage, you may create additional indexes to support the application.

Performance problems due to contention often involve inserts into the same block or incorrect use of sequence numbers. Use particular care in the design, use, and location of indexes, as well as in using the sequence generator and clusters.

> **See Also:** For more information, see "Using Indexes" in Chapter 12, "Data Access Methods".

Step 5: Tune Database Operations

Before tuning the Oracle server, be certain that your application is taking full advantage of the SQL language and the Oracle features designed to enhance application processing. Use features and techniques such as the following, based on the needs of your application:

- Array processing
- The Oracle optimizer
- The row-level lock manager
- PL/SQL

Understanding Oracle's query processing mechanisms is also important for writing effective SQL statements.

> **See Also:** Part II, "Application Design Tuning for Designers and Developers" discusses the Oracle optimizer and how to write statements to achieve optimal performance. It also discusses statistics management and describes preserving execution plans with the plan stability feature.

Whether you are writing new SQL statements or tuning problematic statements in an existing application, your methodology for tuning database operations essentially concerns CPU and disk I/O resources.

- Step 1: Find the Statements that Consume the Most Resources
- Step 2: Tune These Statements To Use Fewer Resources

Step 1: Find the Statements that Consume the Most Resources

Focus your tuning efforts on statements where the benefit of tuning demonstrably exceeds the cost of tuning. Use tools such as `TKPROF`, the SQL trace facility, SQL Analyze, Oracle Trace, and the Enterprise Manager Tuning Pack to find the problem statements and stored procedures. Alternatively, you can query the `V$SORT_USAGE` view to see the session and SQL statement associated with a temporary segment.

The statements with the most potential to improve performance, if tuned, include:

- Those consuming greatest resource overall.
- Those consuming greatest resource per row.
- Those executed most frequently.

In the `V$SQLAREA` view, you can find those statements still in the cache that have done a great deal of disk I/O and buffer gets. (Buffer gets show approximately the amount of CPU resource used.)

> **See Also:** For more information on dynamic performance views, see Chapter 6, "Using SQL Trace and TKPROF", Chapter 14, "Using Oracle Trace", and *Oracle8i Reference.*

Step 2: Tune These Statements To Use Fewer Resources

Remember that application design is fundamental to performance. No amount of SQL statement tuning can make up for inefficient application design. If you

encounter SQL statement tuning problems, then perhaps you need to change the application design.

You can use two strategies to reduce the resources consumed by a particular statement:

- Get the statement to use fewer resources.
- Use the statement less frequently.

Statements may use more resources because they do the most work, or because they perform their work inefficiently—or they may do both. However, the lower the resource used per unit of work (per row processed), the more likely it is that you can significantly reduce resources used only by changing the application itself. That is, rather than changing the SQL, it may be more effective to have the application process fewer rows, or process the same rows less frequently.

These two approaches are not mutually exclusive. The former is clearly less expensive, because you should be able to accomplish it either without program change (by changing index structures) or by changing only the SQL statement itself rather than the surrounding logic.

> **See Also:** For more information, see Chapter 18, "Tuning CPU Resources" and Chapter 20, "Tuning I/O".

Step 6: Tune the Access Paths

Ensure that there is efficient data access. Consider the use of clusters, hash clusters, B*-tree indexes, bitmap indexes, and optimizer hints. Also consider analyzing tables and using histograms to analyze columns in order to help the optimizer determine the best query plan.

Ensuring efficient access may mean adding indexes or adding indexes for a particular application and then dropping them again. It may also mean re-analyzing your design after you have built the database. You may want to further normalize your data or create alternative indexes. Upon testing the application, you may find that you are still not obtaining the required response time. If this happens, then look for more ways to improve the design.

> **See Also:** Chapter 12, "Data Access Methods".

Step 7: Tune Memory Allocation

Appropriate allocation of memory resources to Oracle memory structures can have a positive effect on performance.

Oracle8*i* shared memory is allocated dynamically to the following structures, which are all part of the shared pool. Although you explicitly set the total amount of memory available in the shared pool, the system dynamically sets the size of each of the following structures contained within it:

- The data dictionary cache
- The library cache
- Context areas (if running a multi-threaded server)

You can explicitly set memory allocation for the following structures:

- Buffer cache
- Log buffer
- Sequence caches

Proper allocation of memory resources improves cache performance, reduces parsing of SQL statements, and reduces paging and swapping.

Process local areas include:

- Context areas (for systems not running a multi-threaded server)
- Sort areas
- Hash areas

Be careful not to allocate to the system global area (SGA) such a large percentage of the machine's physical memory that it causes paging or swapping.

> **See Also:** For more information on memory structures and processes, see Chapter 19, "Tuning Memory Allocation" and *Oracle8i Concepts.*

Step 8: Tune I/O and Physical Structure

Disk I/O tends to reduce the performance of many software applications. The Oracle server, however, is designed so that its performance is not unduly limited by I/O. Tuning I/O and physical structure involves these procedures:

- Distributing data so that I/O is distributed to avoid disk contention.
- Storing data in data blocks for best access: setting an adequate number of free lists and using proper values for `PCTFREE` and `PCTUSED`.

- Creating extents large enough for your data, to avoid dynamic extension of tables. This adversely affects the performance of high-volume OLTP applications.
- Evaluating the use of raw devices.

See Also: Chapter 20, "Tuning I/O".

Step 9: Tune Resource Contention

Concurrent processing by multiple Oracle users may create contention for Oracle resources. Contention may cause processes to wait until resources are available. Take care to reduce the following types of contention:

- Block contention
- Shared pool contention
- Lock contention
- Pinging (in a parallel server environment)
- Latch contention

See Also: Chapter 21, "Tuning Resource Contention".

Step 10: Tune the Underlying Platform(s)

See your platform-specific Oracle documentation for ways to tune the underlying system. For example, on UNIX-based systems you might want to tune the following:

- Size of the UNIX buffer cache
- Logical volume managers
- Memory and size for each process

See Also: Chapter 23, "Tuning the Operating System".

Applying the Tuning Method

This section explains how to apply the tuning method:

- Set Clear Goals for Tuning
- Create Minimum Repeatable Tests
- Test Hypotheses
- Keep Records and Automate Testing
- Avoid Common Errors
- Stop Tuning When Objectives Are Met
- Demonstrate Meeting the Objectives

Set Clear Goals for Tuning

Never begin tuning without having first established clear objectives: you cannot succeed without a definition of "success."

"Just make it go as fast as you can" may sound like an objective, but it is very difficult to determine whether this has been achieved. It is even more difficult to tell whether your results have met the underlying business requirements. A more useful objective is: "We need to have as many as 20 operators, each entering 20 orders per hour, and the packing lists must be produced within 30 minutes of the end of the shift."

Keep your goals in mind as you consider each tuning measure. Consider its performance benefits in light of your goals.

Also remember that your goals may conflict. For example, to achieve best performance for a specific SQL statement, you may need to sacrifice the performance of other SQL statements running concurrently on your database.

Create Minimum Repeatable Tests

Create a series of minimum repeatable tests. For example, if you identify a single SQL statement that is causing performance problems, then run both the original and the revised version of that statement in SQL*Plus (with the SQL Trace Facility or Oracle Trace enabled), so that you can see statistically the difference in performance. In many cases, a tuning effort can succeed simply by identifying one SQL statement that was causing the performance problem.

For example, assume that you need to reduce a 4-hour run to 2 hours. To do this, perform your trial runs using a test environment similar to the production environment. For example, you could impose additional restrictive conditions, such as processing one department instead of all 500 departments. The ideal test case should run for more than 1 minute but probably not longer than 5, so you can intuitively detect improvements. You should also measure the test run using timing features.

Test Hypotheses

With a minimum repeatable test established, and with a script both to conduct the test and to summarize and report the results, you can test various hypotheses to see the effect.

Remember that with Oracle's caching algorithms, the first time data is cached there is more overhead than when the same date is later accessed from memory. Thus, if you perform two tests, one after the other, then the second test should run faster then the first. This is because data that the test run would otherwise have had to read from disk may instead be more quickly retrieved from the cache.

Keep Records and Automate Testing

Keep records of the effect of each change by incorporating record keeping into the test script. You also should automate testing. Automation provides a number of advantages:

- It permits cost effectiveness in terms of the tuner's ability to conduct tests quickly.
- It helps ensure that tests are conducted in the same systematic way, using the same instrumentation for each hypothesis you are testing.

You should also carefully check test results derived from observations of system performance against the objective data before accepting them.

Avoid Common Errors

A common error made by inexperienced tuners is to adhere to preconceived notions about what may be causing the problem. The next most common error is to attempt various solutions at random.

Scrutinize your resolution process by developing a written description of your theory of what you think the problem is. This often helps you detect mistakes, simply from articulating your ideas. For best results, consult a team of people to

help resolve performance problems. While a performance tuner can tune SQL statements without knowing the application in detail, the team should include someone who understands the application and who can validate the solutions the SQL tuner may devise.

Avoid Poorly Thought Out Solutions

Beware of changing something in the system by guessing. Or, once you have a hypothesis that you have not completely thought through, you may be tempted to implement it globally. Doing this in haste can seriously degrade system performance to the point where you may have to rebuild part of your environment from backups.

Avoid Preconceptions

Try to avoid preconceptions when you address a tuning problem. Ask users to describe performance problems. However, do not expect users to know why the problem exists.

One user, for example, had serious system memory problems over a long period of time. During the morning, the system ran well, but performance rapidly degraded in the afternoon. A consultant tuning the system was told that a PL/SQL memory leak was the cause. As it turned out, this was not at all the problem.

Instead, the user had set `SORT_AREA_SIZE` to 10MB on a machine with 64 MB of memory serving 20 users. When users logged on to the system, the first time they executed a sort, their sessions were assigned to a sort area. Each session held the sort area for the duration of the session. So, the system was burdened with 200MB of virtual memory, hopelessly swapping and paging.

Stop Tuning When Objectives Are Met

One of the great advantages of having targets for tuning is that it becomes possible to define success. Past a certain point, it is no longer cost effective to continue tuning a system.

Demonstrate Meeting the Objectives

As the tuner, you may be confident that performance targets have been met. Nonetheless, you must demonstrate this to two communities:

- The users affected by the problem.
- Those responsible for the application's success.

Part II

Application Design Tuning for Designers and Developers

Part II provides information on designing and tuning applications for optimal performance. The chapters in Part II are:

- Chapter 3, "Application and System Performance Characteristics"
- Chapter 4, "The Optimizer"
- Chapter 5, "Using EXPLAIN PLAN"
- Chapter 6, "Using SQL Trace and TKPROF"
- Chapter 7, "Using Optimizer Hints"
- Chapter 8, "Gathering Statistics"
- Chapter 9, "Optimizing SQL Statements"
- Chapter 10, "Using Plan Stability"

3 Application and System Performance Characteristics

This chapter describes types of applications and systems that use Oracle databases, and the suggested approaches and features available when designing each type.

This chapter contains the following sections:

- Types of Applications
- Registering Applications
- Oracle Configurations

Types of Applications

You can build thousands of types of applications on top of an Oracle Server. This section categorizes the most popular types and describes the design considerations for each. Each category lists performance issues that are crucial for that type of application.

- Online Transaction Processing (OLTP)
- Decision Support Systems
- Multipurpose Applications

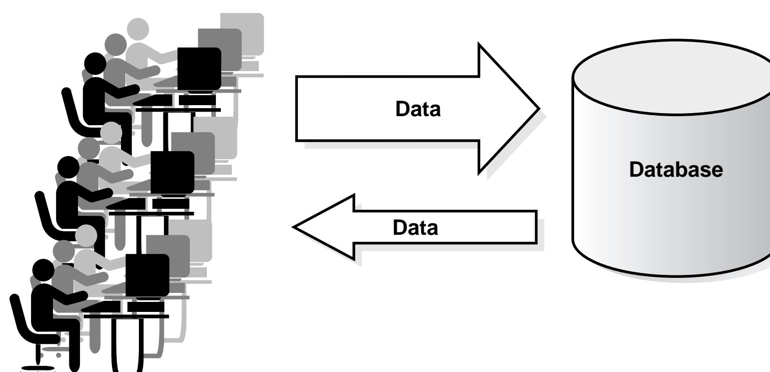
See Also: For more information on these topics and how to implement their features, see *Oracle8i Concepts, Oracle8i Application Developer's Guide - Fundamentals,* and *Oracle8i Administrator's Guide*.

Online Transaction Processing (OLTP)

Online transaction processing (OLTP) applications are high throughput and insert/update-intensive. These applications are characterized by growing volumes of data that several hundred users access concurrently. Typical OLTP applications are airline reservation systems, large order-entry applications, and banking applications. The key goals of OLTP applications are availability (sometimes 7 day/24 hour availability); speed (throughput); concurrency; and recoverability.

Figure 3–1 illustrates the interaction between an OLTP application and an Oracle Server.

Figure 3–1 Online Transaction Processing Systems



When you design an OLTP system, you must ensure that the large number of concurrent users does not interfere with the system's performance. You must also avoid excessive use of indexes and clusters, because these structures slow down insert and update activity.

The following elements are crucial for tuning OLTP systems:

- Rollback segments
- Indexes, clusters, and hashing
- Discrete transactions
- Data block size
- Buffer cache size
- Dynamic allocation of space to tables and rollback segments
- Transaction processing monitors and the multi-threaded server
- Use of bind variables
- The shared pool
- Partitioning
- Well-tuned SQL statements
- Integrity constraints
- Client/server architecture
- Dynamically changeable initialization parameters

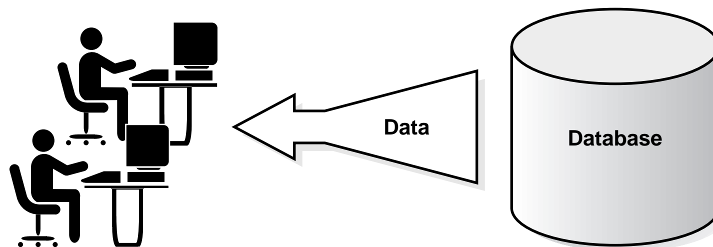
- Procedures, packages, and functions

> **See Also:** For descriptions of these topics, see *Oracle8i Concepts* and *Oracle8i Administrator's Guide*. Read more about these topics before designing your system, and decide which features can benefit your particular situation.

Decision Support Systems

Decision support systems applications typically convert large amounts of information into user-defined reports. Decision support applications perform queries on the large amounts of data gathered from OLTP applications. Decision makers use these applications to determine what strategies the organization should take. Figure 3–2 illustrates the interaction between a decision support application and an Oracle Server.

Figure 3–2 Decision Support Systems



An example of a decision support system is a marketing tool that determines the buying patterns of consumers based on information gathered from demographic studies. The demographic data is assembled and entered into the system, and the marketing staff queries this data to determine which items sell best in which locations. This report helps users decide which items to purchase and market in the various locations.

The key goals of a decision support system are response time, accuracy, and availability. When designing decision support systems, ensure that queries on large amounts of data are performed within a reasonable timeframe. Decision makers often need reports on a daily basis, so you may need to guarantee that the report completes overnight.

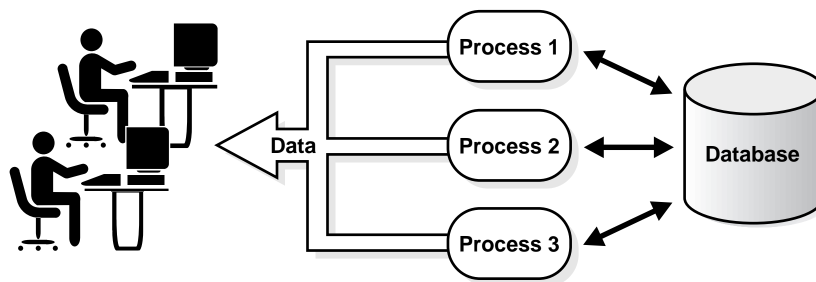
The key to performance in a decision support system is properly tuned queries and proper use of indexes, clusters, and hashing. The following issues are crucial in implementing and tuning a decision support system:

- Materialized Views
- Indexes (B*-tree and bitmap)
- Clusters, hashing
- Data block size
- Parallel execution
- Star query
- The optimizer
- Using hints in queries
- PL/SQL functions in SQL statements
- Partitioning

One way to improve the response time in decision support systems is to use parallel execution. This feature enables multiple processes to simultaneously process a single SQL statement. By spreading processing over many processes, Oracle can execute complex statements more quickly than if only a single server processed them.

Figure 3–3 illustrates parallel execution.

Figure 3–3 Parallel Execution Processing



Parallel execution can dramatically improve performance for data-intensive operations associated with decision support applications or very large database environments. In some cases, it can also benefit OLTP processing.

Symmetric multiprocessing (SMP), clustered, or massively parallel systems gain the largest performance benefits from parallel execution. This is because operations can be effectively spread among many CPUs on a single system.

Parallel execution helps system performance scale when adding hardware resources. If your system's CPUs and disk controllers are already heavily loaded, then reduce the system's load before attempting to use parallel execution to improve performance.

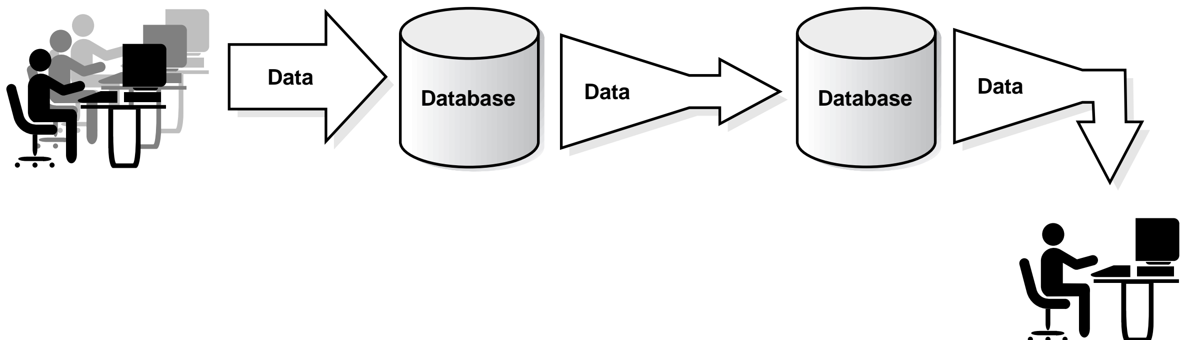
> **See Also:** For more information on data warehousing and parallel execution, see *Oracle8i Data Warehousing Guide.* For general information on parallel execution, see *Oracle8i Concepts.*

Multipurpose Applications

Many applications rely on several configurations. You must decide what type of activity your application performs and determine which features are best suited for it. One typical multipurpose configuration is a combination of OLTP and data warehousing systems. Often, data gathered by an OLTP application "feeds" a data warehousing system.

Figure 3–4 illustrates multiple configurations and applications accessing an Oracle Server.

Figure 3–4 A Hybrid OLTP/Data Warehousing System



One example of a combination OLTP/data warehousing system is a marketing tool that determines the buying patterns of consumers based on information gathered from retail stores. The retail stores gather data from daily purchase records, and the marketing staff queries this data to determine which items sell best in which

locations. This report is then used to determine inventory levels for particular items in each store.

In this example, both systems could use the same database, but the conflicting goals of OLTP and data warehousing might cause performance problems. To solve this, an OLTP database stores the data gathered by the retail stores, then an image of that data is copied into a second database, which is queried by the data warehousing application. This configuration may slightly compromise the goal of accuracy for the data warehousing application (the data is copied only once per day), but the benefit is significantly better performance from both systems.

For hybrid systems, determine which goals are most important. You may need to compromise on meeting lower-priority goals to achieve acceptable performance across the whole system.

Registering Applications

Application developers can use the `DBMS_APPLICATION_INFO` package with Oracle Trace and the SQL trace facility to register the name of the application and actions performed by that application with the database. Registering an application lets system administrators and performance tuning specialists track performance by module. System administrators can also use this information to track resource use by module. When an application registers with the database, its name and actions are recorded in the `V$SESSION` and `V$SQLAREA` views.

Your applications should set the name of the module and name of the action automatically each time a user enters that module. The module name could be the name of a form in an Oracle Developer application, or the name of the code segment in an Oracle precompilers application. The action name should usually be the name or description of the current transaction within a module.

> **See Also:** For information about the required privileges and the procedures in `DBMS_APPLICATION_INFO`, see *Oracle8i Supplied PL/SQL Packages Reference.*

Oracle Configurations

You can configure your system depending on the hardware and software available. The basic configurations are:

- Distributed Systems
- Multi-Tier Systems

- Oracle Parallel Server
- Client/Server Configurations

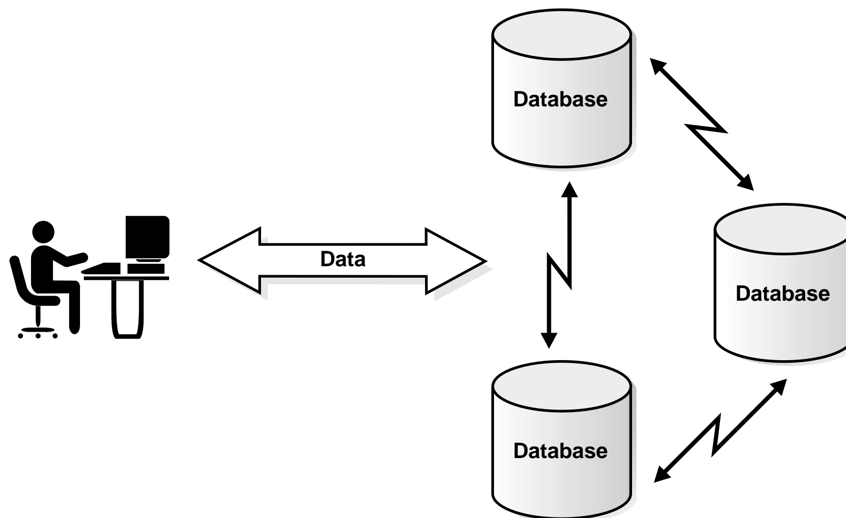
Depending on your application and your operating system, each of these, or a combination of these, configurations may best suit your needs.

Distributed Systems

Distributed applications spread data over multiple databases on multiple machines. Several smaller server machines can be less expensive and more flexible than one large, centrally located server. Distributed configurations take advantage of small, powerful server machines and less expensive connectivity options. Distributed systems also allow you to store data at several sites, and each site can transparently access all the data.

Figure 3–5 illustrates the distributed database configuration of the Oracle Server.

Figure 3–5 Distributed Database System



An example of a distributed database system is a mail order application with order entry clerks in several locations across the country. Each clerk has access to a copy of the central inventory database, but clerks also perform local operations on a local order-entry system. The local orders are forwarded daily to the central shipping department. The local order-entry system is convenient for clerks serving customers

in the same geographic region. The centralized nature of the company-wide inventory database provides processing convenience for the mail order function.

The key goals of a distributed database system are availability, accuracy, concurrency, and recoverability. When you design a distributed system, the location of the data is the most important factor. You must ensure that local clients have quick access to the data they use most frequently. You must also ensure that remote operations do not occur often. Replication is one means of dealing with the issue of data location. The following issues are crucial to the design of distributed database systems:

- Network configuration
- Distributed database design
- Symmetric replication
- Table snapshots and snapshot logs
- Procedures, packages, and functions

> **See Also:** For more information on distributed queries, see *Oracle8i Distributed Database Systems, Oracle8i Replication,* and Chapter 9, "Optimizing SQL Statements".

Multi-Tier Systems

A *multi-tier architecture* has the following components:

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

This architecture allows you to use an application server to do the following:

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

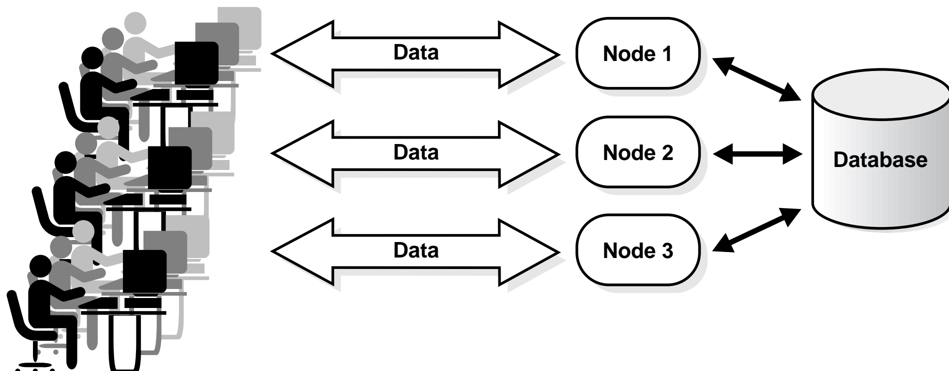
The identity of the client is maintained throughout all tiers of the connection. The Oracle database server audits operations that the application server performs on behalf of the client separately from operations that the application server performs on its own behalf (such as a request for a connection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

See Also: For more information on multi-tier systems, see "Solving CPU Problems by Changing System Architectures" on page 18-13.

Oracle Parallel Server

The Oracle Parallel Server is available on clustered or massively parallel systems. A parallel server allows multiple machines to have separate instances access the same database. This configuration greatly enhances data throughput. Figure 3–6 illustrates the Oracle Parallel Server.

Figure 3–6 An Oracle Parallel Server



When configuring Oracle Parallel Server, a key concern is preventing data contention among the various nodes. Although the cache fusion feature of Oracle Parallel Server minimizes block pinging among nodes contending for data, you should still strive to properly partition data. This is especially true for write/write conflicts where each node must first obtain a lock on that data to ensure data consistency.

If multiple nodes require access to the same data for DML operations, then that data must first be written to disk before the next node can obtain a lock. This type of

contention significantly degrades performance. On such systems, data must be effectively partitioned among the various nodes for optimal performance. Read-only data can be efficiently shared across all instances in an Oracle Parallel Server configuration without the problem of lock contention, because Oracle uses a non-locking query logic. Consider adding sufficient free lists on tables that are mostly inserted.

> **See Also:** For more information, see the *Oracle8i Parallel Server Documentation Set: Oracle8i Parallel Server Concepts; Oracle8i Parallel Server Setup and Configuration Guide; Oracle8i Parallel Server Administration, Deployment, and Performance.*

Client/Server Configurations

Client/server architectures distribute the work of a system between the client (application) machine and the server (in this case an Oracle Server). Typically, client machines are workstations that execute a graphical user interface (GUI) application connected to a larger server machine that houses the Oracle Server.

4

The Optimizer

This chapter discusses SQL processing, optimization methods, and how the optimizer chooses to execute SQL statements.

This chapter contains the following sections:

- SQL Processing Architecture
- EXPLAIN PLAN
- What Is The Optimizer?
- Choosing an Optimizer Approach and Goal
- Cost-Based Optimizer (CBO)
- CBO Parameters
- Extensible Optimizer
- Rule-Based Optimizer (RBO)
- Overview of Optimizer Operations
- Optimizing Joins
- Optimizing Statements that Use Common Subexpressions
- Evaluation of Expressions and Conditions
- Transforming and Optimizing Statements

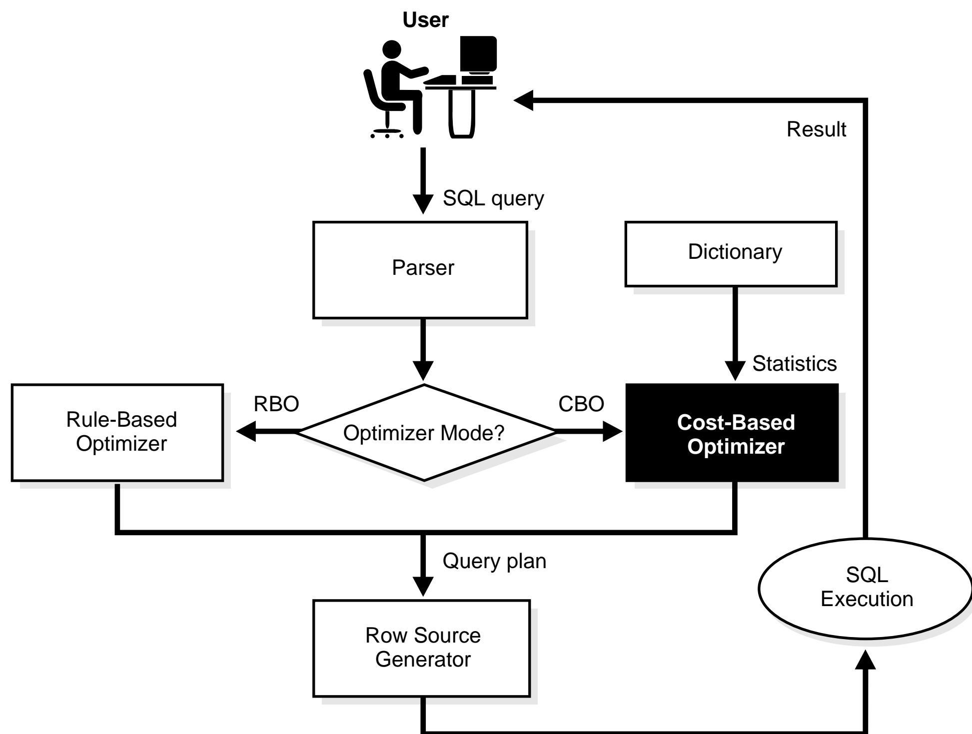
SQL Processing Architecture

The SQL processing architecture is comprised of the following main components:

- Parser
- Optimizer
- Row Source Generator
- SQL Execution

Figure 4–1 illustrates the SQL processing architecture:

Figure 4–1 SQL Processing Architecture



The parser, the optimizer, and the row source generator form the *SQL Compiler*. This compiles the SQL statements into a shared cursor. Associated with the shared cursor is the execution plan.

Parser

The parser performs two functions:

- Syntax analysis: This checks SQL statements for correct syntax.
- Semantic analysis: This checks, for example, that the current database objects and object attributes referenced are correct.

Optimizer

The optimizer is the heart of the SQL processing engine. The Oracle server provides two methods of optimization: rule-based optimizer (RBO) and cost-based optimizer (CBO).

Row Source Generator

The row source generator receives the optimal plan from the optimizer. It outputs the execution plan for the SQL statement. The execution plan is a collection of row sources structured in the form of a tree. A *row source* is an iterative control structure. It processes a set of rows, one row at a time, in an iterated manner. A row source produces a row set.

SQL Execution

SQL execution is the component that operates on the execution plan associated with a SQL statement. It then produces the results of the query.

EXPLAIN PLAN

You can examine the execution plan chosen by the optimizer for a SQL statement by using the `EXPLAIN PLAN` statement. This causes the optimizer to choose the execution plan, and then insert data describing the plan into a database table.

Simply issue the `EXPLAIN PLAN` statement and then query the output table. The following output table describes 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 4–2 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 4–2.

> **See Also:** For detailed information on how to use EXPLAIN PLAN and how to produce and interpret its output, see Chapter 5, "Using EXPLAIN PLAN".

What Is The Optimizer?

The optimizer determines the most efficient way to execute a SQL statement. This is an important step in the processing of any data manipulation language (DML) statement: SELECT, INSERT, UPDATE, or DELETE. There are often many different ways to execute a SQL statement; for example, by varying the order in which tables or indexes are accessed. The procedure Oracle uses to execute a statement can greatly affect how quickly the statement executes.

The optimizer considers many factors among alternative access paths. It can use either a a cost-based or a rule-based approach (see "Cost-Based Optimizer (CBO)" on page 4-12 and "Rule-Based Optimizer (RBO)" on page 4-34).

> **Note:** The optimizer may not make the same decisions from one version of Oracle to the next. In recent versions, the optimizer may make different decisions based on better information available to it.

You can influence the optimizer's choices by setting the optimizer approach and goal, and by gathering statistics for the CBO. Sometimes, the application designer, who has more information about a particular application's data than is available to the optimizer, can choose a more effective way to execute a SQL statement. The application designer can use hints in SQL statements to specify how the statement should be executed.

See Also:

- For more information on optimization goals, see "Choosing an Optimizer Approach and Goal" on page 4-8.
- For more information on using statistics, see Chapter 8, "Gathering Statistics".
- For more information about using hints in SQL statements, see Chapter 7, "Using Optimizer Hints".

Execution Plan

To execute a DML statement, Oracle may need to perform many steps. Each of these steps either retrieves rows of data physically from the database or prepares them in some way for the user issuing the statement. The combination of the steps Oracle uses to execute a statement is called an *execution plan*. An execution plan includes an *access method* for each table that the statement accesses and an ordering of the tables (the *join order*).

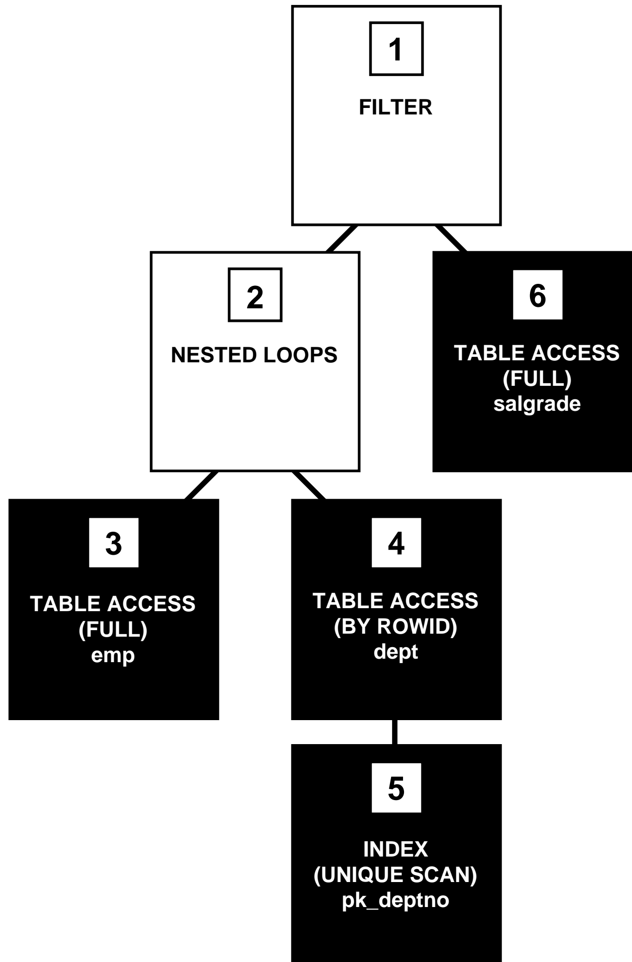
See Also: For descriptions of the various access methods, including indexes, hash clusters, and table scans, see "Access Paths for the RBO" on page 4-34 and "Access Paths for the CBO" on page 4-20.

The following SQL statement 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 4–2 shows a graphical representation of the execution plan for this SQL statement.

Figure 4–2 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 4–2 is a hierarchical diagram showing the flow of row sources from one step to another. The numbering of the steps reflects the order in which they are

displayed in response to the EXPLAIN PLAN statement. Generally, this is *not* the order in which the steps are executed.

> **See Also:** EXPLAIN PLAN is described in "EXPLAIN PLAN" section below. The order in which the steps are executed is described in "Execution Order" on page 4-7.

Each step of the execution plan either retrieves rows from the database or accepts rows from one or more row sources as input:

- Steps indicated by the shaded boxes physically retrieve data from an object in the database. Such steps are called *access paths*:
  - Steps 3 and 6 read all the rows of the `emp` and `salgrade` tables, respectively.
  - Step 5 looks up each `deptno` value in the `pk_deptno` index returned by step 3. There it finds the rowids of the associated rows in the `dept` table.
  - Step 4 retrieves the rows whose rowids were returned by step 5 from the `dept` table.
- Steps indicated by the clear boxes operate on row sources:
  - Step 2 performs a nested loops operation, accepting row sources from steps 3 and 4, joining each row from step 3 source to its corresponding row in step 4, and returning the resulting rows to step 1.
  - Step 1 performs a filter operation. It accepts row sources from steps 2 and 6, eliminates rows from step 2 that have a corresponding row in step 6, and returns the remaining rows from step 2 to the user or application issuing the statement.

> **See Also:** For more information on access paths, see "Access Paths for the RBO" on page 4-34 and "Access Paths for the CBO" on page 4-20. For more information on the methods by which Oracle joins row sources, see "Optimizing Joins" on page 4-49.

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 4–2). The rows returned by each step become the row sources of its parent step. Then, Oracle performs the parent steps.

For example, Oracle performs the following steps to execute the statement in Figure 4–2:

- Oracle performs step 3 and returns the resulting rows, one by one, to step 2.
- For each row returned by step 3, Oracle performs the following 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 returns 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, then 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, then 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, then Oracle cannot perform the parent step until all rows have been returned from the child step. Such parent steps include sorts, sort-merge joins, and aggregate functions.

Choosing an Optimizer Approach and Goal

By default, the goal of the CBO is the best *throughput*; i.e., using the least amount of resources necessary to process all rows accessed by the statement.

Oracle can also optimize a statement with the goal of best *response time*; i.e., using the least amount of resources necessary to process the first row accessed by a SQL statement.

For parallel execution of a SQL statement, the optimizer can choose to minimize elapsed time at the expense of resource consumption. The initialization parameter `OPTIMIZER_PERCENT_PARALLEL` specifies how much the optimizer attempts to parallelize execution.

The execution plan produced by the optimizer can vary depending on the optimizer's goal. Optimizing for best throughput is more likely to result in a full table scan rather than an index scan, or a sort-merge join rather than a nested loops join. Optimizing for best response time, however, more likely results in an index scan or a nested loops join.

For example, suppose you have a join statement that is executable with either a nested loops operation or a sort-merge operation. The sort-merge operation may return the entire query result faster, while the nested loops operation may return the first row faster. If your goal is to improve throughput, then the optimizer is more likely to choose a sort-merge join. If your goal is to improve response time, then the optimizer is more likely to choose a nested loops join.

Choose a goal for the optimizer based on the needs of your application:

- For applications performed in batch, such as Oracle Reports applications, optimize for best throughput. Throughput is usually more important in batch applications, because the user initiating the application is only concerned with the time necessary for the application to complete. Response time is less important, because the user does not examine the results of individual statements while the application is running.
- For interactive applications, such as Oracle Forms applications or SQL*Plus queries, optimize for best response time. Response time is usually important in interactive applications, because the interactive user is waiting to see the first row accessed by the statement.
- For queries that use `ROWNUM` to limit the number of rows, optimize for best response time. Because of the semantics of `ROWNUM` queries, optimizing for response time provides the best results.

The optimizer's behavior when choosing an optimization approach and goal for a SQL statement is affected by the following factors:

- OPTIMIZER_MODE Initialization Parameter
- Statistics in the Data Dictionary
- OPTIMIZER_GOAL Parameter of the ALTER SESSION Statement
- Changing the Goal with Hints

OPTIMIZER_MODE Initialization Parameter

The OPTIMIZER_MODE initialization parameter establishes the default behavior for choosing an optimization approach for the instance. It can have the following values:

CHOOSE	The optimizer chooses between a cost-based approach and a rule-based approach based on whether statistics are available for the CBO. If the data dictionary contains statistics for at least one of the accessed tables, then 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, then 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 regardless of the presence of statistics.

If the optimizer uses the cost-based approach for a SQL statement, and if some tables accessed by the statement have no statistics, then the optimizer uses internal information (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 the CBO. You can collect exact or estimated statistics about physical storage characteristics and data distribution in these schema objects by using the DBMS_STATS package, the ANALYZE statement, or the COMPUTE STATISTICS clause of the CREATE or ALTER INDEX statement.

To provide the optimizer with up-to-date statistics, you should collect new statistics after modifying the data or structure of schema objects in ways that could affect their statistics.

See Also: For more information about statistics, see Chapter 8, "Gathering Statistics".

OPTIMIZER_GOAL Parameter of the ALTER SESSION Statement

The OPTIMIZER_GOAL parameter of the ALTER SESSION statement can override the optimizer 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 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, then 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, then 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.

Changing the Goal with 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 statement.

By default, the cost-based approach optimizes for best throughput. You can change the goal of the CBO in the following ways:

- To change the goal of the CBO for *all* SQL statements in your session, issue an `ALTER SESSION SET OPTIMIZER_MODE` statement with the `ALL_ROWS` or `FIRST_ROWS` clause.
- To specify the goal of the CBO for an individual SQL statement, use the `ALL_ROWS` or `FIRST_ROWS` hint.

Example The following statement changes the goal of the CBO for your session to best response time:

```
ALTER SESSION SET OPTIMIZER_MODE = FIRST_ROWS;
```

> **See Also:** For information on how to use hints, see Chapter 7, "Using Optimizer Hints".

Cost-Based Optimizer (CBO)

In general, you should always use the cost-based approach. The rule-based approach is available for the benefit of existing applications.

The CBO determines which execution plan is most efficient by considering available access paths and by factoring in information based on statistics for the schema objects (tables or indexes) accessed by the SQL statement. The CBO also considers hints, which are optimization suggestions placed in a comment in the statement.

> **See Also:** For more information on hints, see Chapter 7, "Using Optimizer Hints".

The CBO consists of the following steps:

1. The optimizer generates a set of potential plans for the SQL statement based on its available access paths and hints.
2. The optimizer estimates the cost of each plan based on statistics in the data dictionary for the data distribution and storage characteristics of the tables, indexes, and partitions accessed by the statement.

   The *cost* is an estimated value proportional to the expected resource use needed to execute the statement with a particular plan. The optimizer calculates the cost of each possible access method and join order based on the estimated computer resources, including (but not limited to) I/O and memory, that are required to execute the statement using the plan.

Serial plans with greater costs take more time to execute than those with smaller costs. When using a parallel plan, however, resource use is not directly related to elapsed time.

3. The optimizer compares the costs of the plans and chooses the one with the smallest cost.

To maintain the effectiveness of the CBO, you *must* gather statistics and keep them current. Gather statistics on your objects using either of the following:

- For releases prior to Oracle8*i*, use the `ANALYZE` statement.
- For Oracle8*i* releases, use the `DBMS_STATS` package.

For table columns which contain skewed data (i.e., values with large variations in number of duplicates), you must collect histograms.

The resulting statistics provide the CBO with information about data uniqueness and distribution. Using this information, the CBO is able to compute plan costs with a high degree of accuracy. This enables the CBO to choose the best execution plan based on the least cost.

> **See Also:** For detailed information on gathering statistics, see Chapter 8, "Gathering Statistics".

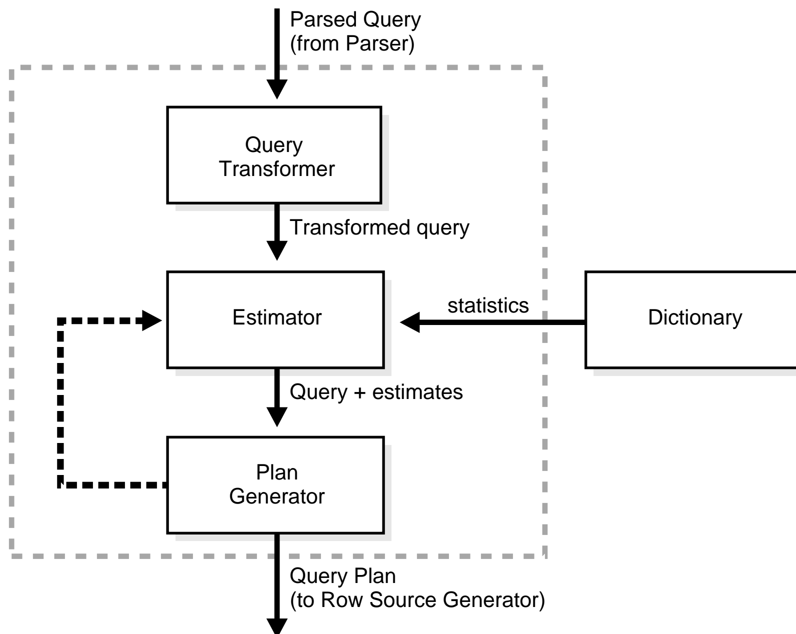
Architecture of the CBO

The CBO consists of the following three main components:

- Query Transformer
- Estimator
- Plan Generator

The CBO architecture is illustrated in Figure 4–3.

Figure 4–3 Cost-Based Optimizer Architecture



Query Transformer

The input to query transformer is a parsed query, which is represented by a set of query blocks. The query blocks are nested or interrelated to each other. The form of the query determines how the query blocks are interrelated to each other. The main objective of the query transformer is to determine if it is advantageous to change the form of the query, so that it enables generation of a better query plan. Three different query transformation techniques are employed by the query transformer: view merging, subquery unnesting, and query rewrite using materialized views. Any combination of these transformations may be applied to a given query.

View Merging Each view referenced in a query is expanded by the parser into a separate query block. The query block essentially represents the view definition, and therefore the result of a view. One option for the optimizer is to optimize the view query block separately, and generate a subplan. Then, optimize the rest of the query by using the view subplan in the generation of overall query plan. Doing so usually leads to a sub-optimal query plan, because the view is optimized separately from rest of the query.

The query transformer removes the potential sub-optimality by merging the view query block into the query block that contains the view. Most of the views are merged, with an exception of few types of views. When a view is merged, the query block representing the view is merged into the containing query block. Now, there is no need to generate a subplan, because view query block is eliminated.

For those views that are not merged, the query transformer pushes the relevant predicates from the containing query block into the view query block. Doing so improves the subplan of the non-merged view, because the pushed in predicates act either as index drivers or as filters.

Subquery Unnesting Like a view, a subquery is also represented by a separate query block. Because a subquery is nested within the main query or another subquery, this constrains the plan generator in trying out different possible plans before it finds a plan with the lowest cost. For this reason, the query plan produced may not be the optimal one. The restrictions due to the nesting of subqueries can be removed by unnesting the subqueries and converting them into joins. Most of the subqueries are unnested. For those subqueries that remain as nested subqueries, separate subplans are generated. To improve the execution speed of the overall query plan, the subplans are ordered in an efficient manner.

> **See Also:** For more information on subquery unnesting, see "Use Care When Unnesting Subqueries" in Chapter 9, "Optimizing SQL Statements".

Query Rewrite with Materialized Views A materialized view is like a query whose result is materialized and stored in a table. When a user query is found compatible with the query associated with a materialized view, the user query can be rewritten in terms of the materialized view. Doing so improves the execution of the user query, because most of the query result has already been precomputed. The query transformer looks for any materialized views that are compatible with the user query, and selects one or more materialized views to rewrite the user query. The use of materialized views to rewrite a query is cost-based. That is, the query is not rewritten if the plan generated without the materialized views has lower cost than the plan generated with the materialized views.

> **See Also:** For more information on query rewrite, see *Oracle8i Data Warehousing Guide*.

Estimator

The estimator is the heart of the CBO. Its estimates three different types of measures: selectivity, cardinality, and cost. These measures are related to each

other, and one is derived from another. The end goal of the estimator is to estimate the overall cost of a given plan. If statistics are available, then the estimator uses them to compute the measures. The statistics improve the degree of accuracy of the measures.

Selectivity The first type of measure is the selectivity, which represents a fraction of rows from a row set. The row set can be a base table, a view, or the result of a join or a `GROUP BY` operator. The selectivity is tied to a query predicate, such as `last_name` = '`Smith`', or a combination of predicates, such as `last_name` = '`Smith`' `AND` `job_type` = '`Clerk`'. A predicate acts as a filter that filters certain number of rows from a row set. Therefore, the selectivity of a predicate indicates how many rows from a row set will pass the predicate test. The selectivity lies in the value range 0.0 to 1.0. A selectivity of 0.0 means that no rows will be selected from a row set, and a selectivity of 1.0 means that all rows will be selected.

The estimator uses an internal default value for the selectivity if no statistics are available. Different internal defaults are used depending on the predicate type. For example, the internal default for an equality predicate (`last_name` = '`Smith`') is lower than the internal default for a range predicate (`last_name` > '`Smith`'). This is because an equality predicate is expected to usually return a smaller fraction of rows than a range predicate.

When statistics are available, the estimator estimates selectivity based on statistics. For example, for an equality predicate (`last_name` = '`Smith`') the selectivity is set to the reciprocal of the number of distinct values of `last_name`, because the query selects rows that all contain one out of N distinct values. If a histogram is available on the `last_name` column, then the estimator uses it instead of the number of distinct values statistic. The histogram captures the distribution of different values in a column, so its use yields better selectivity estimate. Therefore, having histograms on columns that contain skewed data (i.e., values with large variations in number of duplicates) greatly helps the CBO to generate good plans.

Cardinality Cardinality represents the number of rows in a row set. Here, the row set can be a base table, a view, or the result from a join or `GROUP BY` operator. The base cardinality is the number of rows in a base table. The base cardinality can be captured by analyzing the table. If table statistics are not available, then the estimator uses the number of extents occupied by the table to estimate the base cardinality.

The effective cardinality is the number of rows that will be selected from a base table. The effective cardinality is dependent on the predicates specified on different columns of a base table. This is because each predicate acts as a successive filter on the rows of a base table. The effective cardinality is computed as the product of base

cardinality and combined selectivity of all predicates specified on a table. When there is no predicate on a table, its effective cardinality equals its base cardinality.

The join cardinality is the number of rows produced when two row sets are joined together. A join is a Cartesian product of two row sets with the join predicate applied as a filter to the result. Therefore, the join cardinality is the product of the cardinalities of two row sets, multiplied by the selectivity of the join predicate.

A distinct cardinality is the number of distinct values in a column of a row set. The distinct cardinality of a row set is based on the data in the column. For example, in a row set of 100 rows, if distinct column values are found in 20 rows, then the distinct cardinality is 20.

The group cardinality is the number of rows produced from a row set after the `GROUP BY` operator is applied. The effect of the `GROUP BY` operator is to decrease the number of rows in a row set. The group cardinality depends on the distinct cardinality of each of the grouping columns. For example, if a row set of 100 rows is grouped by `colx`, whose distinct cardinality is 30, then the group cardinality is 30. If the row set of 100 rows is grouped by `colx` and `coly`, and distinct cardinalities of `colx` and `coly` are 30 and 60 respectively, then the group cardinality lies between max(30,60) and 100.

Cost The cost represents units of work or resource used. The CBO uses disk I/O as a unit of work. The other possible work units are cpu and network usage. So, the cost used by the CBO represents an estimate of the number of disk I/Os incurred in performing an operation. The operation can be scanning a table, accessing rows from a table using an index, joining two tables together, or sorting a row set. The cost of a query plan is the number of disk I/Os that are expected to be incurred when the query is executed and its result produced.

The access cost represents the number of units of work done in accessing data from a base table. The access path can be a table scan, a fast full index scan, or an index scan. During table scan or fast full index scan, multiple blocks are read from the disk in a single I/O operation. Therefore, the cost of a table scan or a fast full index scan depends on the number of blocks to scan and the multiblock read count value. The cost for an index scan depends on the levels in the B-tree, the number of index leaf blocks to scan, and the number of rows to fetch using the rowid in the index keys. The cost to fetch rows using rowids depends on the index clustering factor. The higher the clustering factor, the more randomly scattered the individual rows are on the disk. So, a higher clustering factor means it costs more to fetch rows by rowid.

The join cost represents the combination of the individual access costs of the two row sets being joined. In a join, one row set is called inner, and the other is called

outer. In a nested loops join, for every row in the outer row set, the inner row set is accessed to find all matching rows to join. Therefore, in a nested loops join, the inner row set is accessed as many times as the number of rows in the outer row set. The cost of nested loops join = outer access cost + (inner access cost * outer cardinality).

In sort merge join, the two row sets being joined are sorted by the join keys, if they are not already in key order. The cost of sort merge join = outer access cost + inner access cost + sort costs (if sort used).

In hash join, the inner row set is hashed into memory, and a hash table is built using the join key. Then, each row from the outer row set is hashed, and the hash table is probed to join to all matching rows. If the inner row set is very large, then only a portion of it is hashed into memory. This is called a hash partition.

Each row from the outer row set is hashed to probe matching rows in the hash partition. After this, the next portion of the inner row set is hashed into memory, followed by a probe from the outer row set. This process is repeated until all partitions of the inner row set are exhausted. The cost of hash join = (outer access cost * # of hash partitions) + inner access cost.

> **See Also:** For more information on joins, see "Optimizing Joins" on page 4-49.

Plan Generator

The main function of the plan generator is to try out different possible plans for a given query and pick the one that has the lowest cost. Many different plans are possible because of the various combination of different access paths, join methods and join orders that can be used to access and process data in different ways and produce the same result.

A join order is the order in which different join items (such as tables) are accessed and joined together. For example, in a join order of `t1`, `t2`, and `t3`, table `t1` is accessed first. This is followed by access of `t2`, whose data is joined to `t1` data to produce a join of `t1` and `t2`. Finally, `t3` is accessed, and its data is joined to the result of join between `t1` and `t2`.

The plan for a query is established by first generating subplans for each of the unnested subqueries and non-merged views. Each unnested subquery or non-merged view is represented by a separate query block. The query blocks are optimized separately in a bottom-up order. That is, the innermost query block is optimized first, and a subplan is generated for it. The outermost query block, which represents the entire query, is optimized last.

The plan generator explores different plans for a query block by trying out different access paths, join methods, and join orders. The number of possible plans for a query block is proportional to the number of join items in the `FROM` clause. This number rises exponentially with the number of join items.

Because of this reason, the plan generator uses an internal cutoff to reduce the number of plans it tries to find the one with the lowest cost. The cutoff is based on the cost of the current best plan. If current best cost is large, then the plan generator tries harder (i.e., explores more alternate plans) to find a better plan with lower cost. If current best cost is small, then the plan generator ends the search swiftly, because further cost improvement will not be significant.

The cutoff works very well if the plan generator starts with an initial join order that produces a plan with cost close to optimal. Finding a good initial join order is a difficult problem. The plan generator uses a simple heuristic for the initial join order. It orders the join items by their effective cardinalities. The join item with the smallest effective cardinality goes first, and the join item with the largest effective cardinality goes last.

Features that Require the CBO

The use of any of the following features requires the use of the CBO:

- Partitioned tables
- Index-organized tables
- Reverse key indexes
- Function-based indexes
- `SAMPLE` clauses in a `SELECT` statement
- Parallel execution and parallel DML
- Star transformations
- Star joins
- Extensible optimizer
- Query rewrite (materialized views)
- Progress meter
- Hash joins
- Bitmap indexes

- Partition views (release 7.3)

Note: Even if the parameter OPTIMIZER_MODE is set to RULE, the use of these features enables the CBO.

Using the CBO

To use the CBO for a statement, collect statistics for the tables accessed by the statement, and enable the CBO using one of the following methods:

- Make sure that the OPTIMIZER_MODE initialization parameter is set to its default value of CHOOSE.
- To enable the CBO for your session only, issue an ALTER SESSION SET OPTIMIZER_MODE statement with the ALL_ROWS or FIRST_ROWS clause.
- To enable the CBO for an individual SQL statement, use any hint other than RULE.

The plans generated by the CBO depend upon the sizes of the tables, and potentially on the data distributions as well, if histograms are being used. When using the CBO with a small amount of data to test an application prototype, do not assume that the plan chosen for the full-size database will be the same as that chosen for the prototype.

See Also: For information on enabling the CBO, see "CBO Parameters" on page 4-28.

Access Paths for the CBO

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 describes the basic methods by which Oracle can access data.

See Also: For the a list of the access paths that are available for the RBO, as well as their ranking, see "Access Paths for the RBO" on page 4-34.

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.

Sample Table Scans

A sample table scan retrieves a random sample of data from a table. This access method is used when the statement's FROM clause includes the SAMPLE clause or the SAMPLE BLOCK clause. To perform a sample table scan when sampling by rows (the SAMPLE clause), Oracle reads a specified percentage of rows in the table and examines each of these rows to determine whether it satisfies the statement's WHERE clause. To perform a sample table scan when sampling by blocks (the SAMPLE BLOCK clause), Oracle reads a specified percentage of the table's blocks and examines each row in the sampled blocks to determine whether it satisfies the statement's WHERE clause.

Oracle does not support sample table scans when the query involves a join or a remote table. However, you can perform an equivalent operation by using a CREATE TABLE AS SELECT query to materialize a sample of an underlying table and then rewrite the original query to refer to the newly created table sample. Additional queries can be written to materialize samples for other tables. Sample table scans require the CBO.

Example: The following statement uses a sample table scan to access 1% of the emp table, sampling by blocks:

```
SELECT *
    FROM emp SAMPLE BLOCK (1);
```

The EXPLAIN PLAN output for this statement might look like this:

```
OPERATION                 OPTIONS        OBJECT_NAME
-----------------------------------------------------
SELECT STATEMENT
  TABLE ACCESS            SAMPLE         EMP
```

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, then 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, then 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 the following types:

Unique scan	This 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	This can return zero or more rowids, depending on how many rows the statement accesses.
Full scan	This is available if a predicate references one of the columns in the index. The predicate does not need 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 null. Full scan can be used to eliminate a sort operation. It reads the blocks singly.
Fast full scan	This 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 the CBO. You can specify it with the initialization parameter OPTIMIZER_FEATURES_ENABLE or the INDEX_FFS hint. Fast full index scans cannot be performed against bitmap indexes.

Index join

This is a hash join of several indexes that together contain all the columns from the table that are referenced in the query. If an index join is used, then no table access is needed, because all the relevant column values can be retrieved from the indexes. An index join cannot be used to eliminate a sort operation.

Index join is available only with the CBO. You can specify it with the initialization parameter `OPTIMIZER_FEATURES_ENABLE` or the `INDEX_JOIN` hint.

Example: The following statement uses an index join to access the `empno` and `sal` columns, both of which are indexed, in the `emp` table:

```
SELECT empno, sal
    FROM emp
    WHERE sal > 2000;
```

The `EXPLAIN PLAN` output for this statement might look like this:

```
OPERATION                OPTIONS          OBJECT_NAME
---------------------------------------------------------
SELECT STATEMENT
  VIEW                                    index$_join$_001
    HASH JOIN
      INDEX              RANGE SCAN       EMP_SAL
      INDEX              FAST FULL SCAN   EMP_EMPNO
```

Bitmap

This uses a bitmap for key values and a mapping function that converts each bit position to a rowid. Bitmaps can 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 the CBO.

Attention: Bitmap indexes are available only if you have purchased the Oracle8*i* Enterprise Edition. For more information on purchasing options, see *Getting to Know Oracle8i*.

How the CBO Chooses an Access Path

The CBO chooses an access path based on the following 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 (and its `FROM` clause for the `SAMPLE` or `SAMPLE BLOCK` 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. Finally, optimizer chooses the execution plan with the lowest estimated cost.

The optimizer's choice among available access paths can be overridden with hints, except when the statement's `FROM` clause contains `SAMPLE` or `SAMPLE BLOCK`.

> **See Also:** For information about hints in SQL statements, see Chapter 7, "Using Optimizer Hints".

To choose among available access paths, the optimizer considers the following 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. This 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 1: The following query 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, then 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.

Example 2: Consider again the query in the previous example. If the ename column is not a unique or primary key, then 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 3: The following query 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_COL_STATISTICS view (or 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 4: The following query 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. This is an internal default value. The optimizer makes this assumption 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 5: The following query 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 selectivity (an internal default value) for indexed columns in order to favor the use of the index.

Example 6: The following query 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:

$S = ABS(S1 + S2 - 1)$

CBO Parameters

This section contains some, but not all, of the parameters specific to the optimizer. The following sections may be especially useful when tuning Oracle Applications.

Parameters Affecting CBO Plans

The following parameters affect cost-based optimizer plans:

OPTIMIZER_FEATURES_ENABLED	Enables several optimizer features, depending on the user-specified value. For example, if OPTIMIZER_FEATURES_ENABLED=8.1.6, then ALL_ROWS or FIRST_ROWS is also used for the recursive SQL generated by PL/SQL procedures. Prior to release 8.1.6, only CHOOSE or RULE was used for such recursive SQL.
OPTIMIZER_MODE	This initialization parameter sets the mode of the optimizer at instance startup: RULE (use RBO), ALL_ROWS (use CBO for throughput), FIRST_ROWS (use CBO for response time), or CHOOSE (an optimizer choice based on the presence of statistics). Set the OPTIMIZER_MODE parameter of the ALTER SESSION statement to change the value dynamically during a session.
OPTIMIZER_PERCENT_PARALLEL	Defines the amount of parallelism that the optimizer uses in its cost functions.
HASH_AREA_SIZE	Larger values can lower hash join costs, permitting Oracle to perform more hash joins.
SORT_AREA_SIZE	Larger values can lower sort costs, permitting Oracle to perform more sort merge joins.
DB_FILE_MULTIBLOCK_READ_COUNT	Larger values can lower table scan costs and make Oracle favor table scans over indexes.

In data warehousing applications, you often need to set the following parameters:

ALWAYS_ANTI_JOIN	Sets the type of antijoin that Oracle uses: NESTED_LOOPS, MERGE, or HASH.
HASH_JOIN_ENABLED	Enables or disables the hash join feature. This should always be set to true for data warehousing applications.

You rarely need to change the following parameters:

`HASH_MULTIBLOCK_IO_COUNT`	Larger value can lower hash join costs, permitting Oracle to perform more hash joins.
`BITMAP_MERGE_AREA_SIZE`	The size of the area used to merge the different bitmaps that match a range predicate. Larger size favors use of bitmap indexes for range predicates.

See Also: For complete information about each parameter, see *Oracle8i Reference.*

Parameters Affecting How the Optimizer Uses Indexes

The following two parameters address the optimizer's use of indexes for a wide range of statements, particularly nested-loop join statements in both OLTP and DSS applications.

`OPTIMIZER_INDEX_COST_ADJ`	Encourages the use of all indexes, regardless of their selectivity. It also applies to index use in general, rather than to just modeling index caching for nested loops join probes.

OPTIMIZER_INDEX_CACHING	Use this if the following two conditions exist: ■ Indexes Oracle could use for nested loops join probes are frequently cached in your environment. ■ The optimizer is not using nested loops joins aggressively enough. In such an environment, this parameter has two advantages over OPTIMIZER_INDEX_COST_ADJ: First, this parameter favors using selective indexes. If you use a relatively low value for this parameter, then the optimizer effectively models the caches of all non-leaf index blocks. In this case, the optimizer bases the cost of using this index primarily on the basis of its selectivity. Thus, by setting this to a low value, you achieve the desired modeling of the index caching without over-using possibly undesirable indexes that have poor selectivity. Second, the effects of using this parameter are restricted to modeling the use of cached indexes for nested loops join probes. Thus, its use has fewer side effects.

Setting Initialization Parameters

To enable the CBO for Oracle Applications, you must set the following parameters:

- OPTIMIZER_MODE=CHOOSE, FIRST_ROWS, or ALL_ROWS
- OPTIMIZER_FEATURES_ENABLE=8.1.6
- COMPATIBLE=8.1.6

You can set the following parameters to enable additional CBO-related features:

- QUERY_REWRITE_ENABLED=TRUE
- _COMPLEX_VIEW_MERGING=TRUE
- _PUSH_JOIN_PREDICATE=TRUE

Verifying Initialization Parameters

To verify that the initialization parameters have been set correctly, execute the following statement against the dictionary's PARAMETER view:

```
SQL> SELECT NAME, VALUE
     FROM V$PARAMETER
     WHERE NAME LIKE 'optimizer%';
```

This returns the following typical data:

```
NAME                              VALUE
------------------------------    ---------------------
optimizer_features_enable         8.1.6
optimizer_mode                    CHOOSE
optimizer_max_permutations        80000
optimizer_index_cost_adj          100
optimizer_index_caching           0
optimizer_percent_parallel        0
optimizer_search_limit            5
```

Extensible Optimizer

The extensible optimizer is part of the CBO. It allows the authors of user-defined functions and domain indexes to control the three main components that the CBO uses to select an execution plan: statistics, selectivity, and cost evaluation.

The extensible optimizer lets you:

- Associate cost function and default costs with domain indexes, indextypes, packages, and stand-alone functions.
- Associate selectivity function and default selectivity with methods of object types, package functions, and stand-alone functions.
- Associate statistics collection functions with domain indexes and columns of tables.
- Order predicates with functions based on cost.
- Select a user-defined access method (domain index) for a table based on access cost.
- Use the ANALYZE statement to invoke user-defined statistics collection and deletion functions.

- Use new data dictionary views to include information about the statistics collection, cost, or selectivity functions associated with columns, domain indexes, indextypes, or functions.
- Add a hint to preserve the order of evaluation for function predicates.

> **See Also:** For details about the extensible optimizer, see *Oracle8i Data Cartridge Developer's Guide*.

User-Defined Statistics

You can define *statistics collection functions* for domain indexes, individual columns of a table, and user-defined datatypes.

Whenever a domain index is analyzed to gather statistics, Oracle calls the associated statistics collection function. Whenever a column of a table is analyzed, Oracle collects the standard statistics for that column and calls any associated statistics collection function. If a statistics collection function exists for a datatype, then Oracle calls it for each column that has that datatype in the table being analyzed.

User-Defined Selectivity

The selectivity of a predicate in a SQL statement is used to estimate the cost of a particular access method; it is also used to determine the optimal join order. The optimizer cannot compute an accurate selectivity for predicates that contain user-defined operators, because it does not have any information about these operators.

You can define *selectivity functions* for predicates containing user-defined operators, stand-alone functions, package functions, or type methods. The optimizer calls the user-defined selectivity function whenever it encounters a predicate that contains the operator, function, or method in one of the following relations with a constant: <, <=, =, >=, >, or `LIKE`.

User-Defined Costs

The optimizer cannot compute an accurate estimate of the cost of a domain index because it does not know the internal storage structure of the index. Also, the optimizer may underestimate the cost of a user-defined function that invokes PL/SQL, uses recursive SQL, accesses a `BFILE`, or is CPU-intensive.

You can define costs for domain indexes and user-defined stand-alone functions, package functions, and type methods. These user-defined costs can be in the form of default costs that the optimizer simply looks up or they can be full-fledged cost

functions that the optimizer calls to compute the cost.

Rule-Based Optimizer (RBO)

Although Oracle supports the rule-based optimizer, you should design new applications to use the cost-based optimizer. You should also use the CBO for data warehousing applications, because the CBO supports enhanced features for DSS. Many new performance features, such as partitioned tables, improved star query processing, and materialized views, are only available with the CBO.

Note: If you have developed OLTP applications using Oracle version 6, and if you have tuned your SQL statements carefully based on the rules of the optimizer, then you may want to continue using the RBO when you upgrade these applications to a new Oracle release.

If you are using applications provided by third-party vendors, then check with the vendors to determine which type of optimizer is best suited to that application.

If `OPTIMIZER_MODE=CHOOSE`, if statistics do not exist, and if you do not add hints to your SQL statements, then your statements use the RBO. You can use the RBO to access both relational data and object types. If `OPTIMIZER_MODE=FIRST_ROWS` or `ALL_ROWS` and no statistics exist, then the CBO uses default statistics. You should migrate your existing applications to use the cost-based approach.

You can enable the CBO on a trial basis simply by collecting statistics. You can then return to the RBO by deleting the statistics or by setting either the value of the `OPTIMIZER_MODE` initialization parameter or the `OPTIMIZER_MODE` clause of the `ALTER SESSION` statement to `RULE`. You can also use this value if you want to collect and examine statistics for your data without using the cost-based approach.

See Also: For an explanation of how to gather statistics, see Chapter 8, "Gathering Statistics".

Access Paths for the RBO

Using the RBO, the optimizer chooses an execution plan based on the access paths available and the ranks of these access paths. Oracle's ranking of the access paths is heuristic. If there is more than one way to execute a SQL statement, then the RBO

always uses the operation with the lower rank. Usually, operations of lower rank execute faster than those associated with constructs of higher rank.

The access paths and their ranking are listed below:

Path 1: Single Row by Rowid

Path 2: Single Row by Cluster Join

Path 3: Single Row by Hash Cluster Key with Unique or Primary Key

Path 4: Single Row by Unique or Primary Key

Path 5: Clustered Join

Path 6: Hash Cluster Key

Path 7: Indexed Cluster Key

Path 8: Composite Index

Path 9: Single-Column Indexes

Path 10: Bounded Range Search on Indexed Columns

Path 11: Unbounded Range Search on Indexed Columns

Path 12: Sort-Merge Join

Path 13: MAX or MIN of Indexed Column

Path 14: ORDER BY on Indexed Column

Path 15: Full Table Scan

Each of the following sections describes an access path, discusses when it is available, and shows the output generated for it by the EXPLAIN PLAN statement.

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:

```
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 the following 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.

> **See Also:** For information on the nested loops operation, see "Nested Loops (NL) Join" on page 4-50.

Example: In the following statement, 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 the following 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: In the following statement, 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: In the following statement, 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.

> **See Also:** For information on nested loops operations, see "Nested Loops (NL) Join" on page 4-50.

Example: In the following statement, 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: In the following statement, 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. Because all rows with the same cluster key value are stored together, the cluster scan requires only a single rowid to find them all.

Example: In the following statement, 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: In the following statement, 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, then Oracle executes the statement by performing a range scan on the index to retrieve the rowids of the selected rows, and then accesses the table by these rowids.

Example 1: In the following statement, 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, then 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, then 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 2: In the following statement, 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 1: In the following statement, 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 2: In the following statement, 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 the following 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 1: In the following statement, 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 2: In the following statement, 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 3: 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.

> **See Also:** For information on these operations, see "Optimizing Join Statements" on page 4-49.

Example: In the following statement, 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, and all of the following 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. Because only this value is selected, Oracle need not access the table after scanning the index.

Example: In the following statement, 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, and all of the following 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 is 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: In the following statement, 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, except when its `FROM` clause contains `SAMPLE` or `SAMPLE BLOCK`.

Note that the full table scan is the lowest ranked access path on the list. This means that the RBO always chooses an access path that uses an index if one is available, even if a full table scan might execute faster.

The following 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: The following 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
```

Overview of Optimizer Operations

This section describes the types of SQL statements that can be optimized and summarizes the operations performed by the optimizer.

Types of SQL Statements

Oracle optimizes the following 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.
Non-equijoin	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 two or more distinct nodes of a distributed database. A *remote statement* accesses data on one remote node of a distributed database.

Optimizer Operations

For any SQL statement processed by Oracle, the optimizer does the following:

1 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 4-65.)
2 Statement transformation	For complex statements involving, for example, correlated subqueries, the optimizer may transform the original statement into an equivalent join statement. (See "Transforming and Optimizing Statements" on page 4-71.)

3 View merging	For SQL statements that access 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 4-76.)
4 Choice of optimizer approaches	The optimizer chooses either a cost-based or rule-based approach and determines the goal of optimization. (See "Optimizing Joins" on page 4-49.)
5 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 "Access Paths for the CBO" on page 4-20.)
6 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.
7 Choice of join operations	For any join statement, the optimizer chooses an operation to use to perform the join.

Optimizing Joins

This section discusses how the Oracle optimizer executes SQL statements that contain joins, anti-joins, and semi-joins. It also describes how the optimizer can use bitmap indexes to execute star queries, which join a fact table to multiple dimension tables.

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 "Access Paths for the RBO" on page 4-34 and "Access Paths for the CBO" on page 4-20.)

Join Operations	To join each pair of row sources, Oracle must perform one of these operations: ■ Nested Loops (NL) Join ■ Sort-Merge Join ■ Hash Join (not available with the RBO) ■ Cluster Join
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

Nested Loops (NL) Join

To perform a nested loops join, Oracle performs the following 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.

For example, consider table A and B. Each row of B is joined back to A.

For rows 1, 2, 3, ...n-1, n in B, each row in B is joined to each row in A

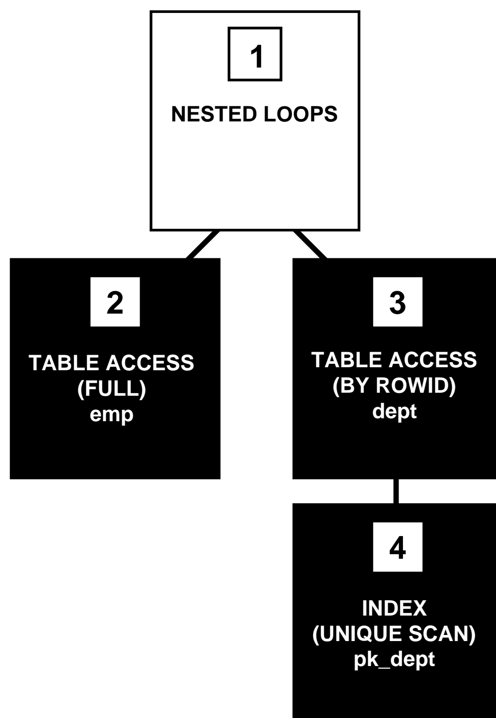
For rows 1, 2, 3, ... n-1, n in A

Total selectivity = selectivity (A) * selectivity (B)

Figure 4–4 shows the execution plan for the following statement using a nested loops join:

```
SELECT *
  FROM emp, dept
  WHERE emp.deptno = dept.deptno;
```

Figure 4–4 Nested Loops Join



To execute this statement, Oracle performs the following 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

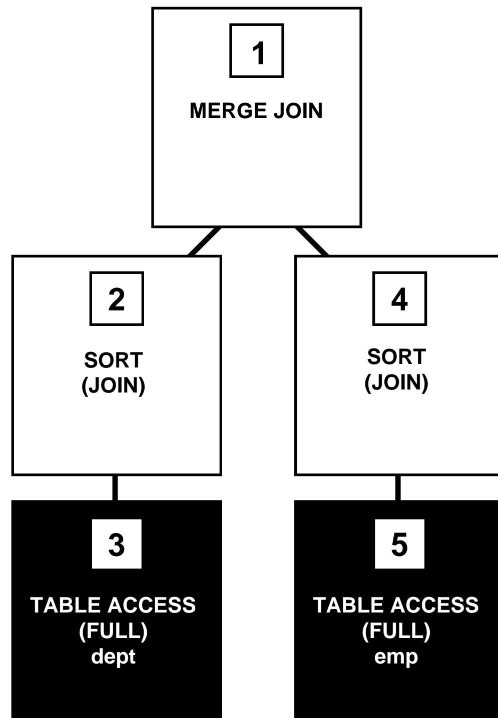
Oracle can only perform a sort-merge join for an equijoin. To perform a sort-merge join, Oracle performs the following steps:

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 4–5 shows the execution plan for this statement using a sort-merge join:

```
SELECT *
  FROM emp, dept
  WHERE emp.deptno = dept.deptno;
```

Figure 4–5 Sort-Merge Join



To execute this statement, Oracle performs the following 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.

Example 2 All relevant table `A` rows are fetched, sorted, and placed in a sort area. The resulting data is:

```
Table A
1
5
8
11
```

All relevant table B rows are fetched, sorted, and placed in a sort area. The resulting data is:

```
Table B
2
4
5
7
```

A merge is then performed using a merge join algorithm to produce the resulting data:

```
Merged Data from A and B
1
2
4
5
7
8
11
```

Hash Join

Oracle can only perform a hash join for an equijoin. Hash join is not available with the RBO. You must enable hash join optimization, using the initialization parameter `HASH_JOIN_ENABLED` (which can be set with the `ALTER SESSION` statement) or the `USE_HASH` hint.

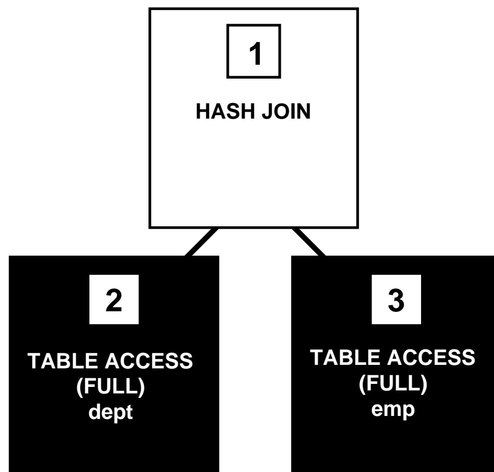
To perform a hash join, Oracle performs the following steps:

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 selects 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 4–6 shows the execution plan for this statement using a hash join:

```
SELECT *
  FROM emp, dept
  WHERE emp.deptno = dept.deptno;
```

Figure 4–6 Hash Join



To execute this statement, Oracle performs the following 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.

> **See Also:** For more information about the USE_HASH hint, see Chapter 7, "Using Optimizer Hints".

Example 2 Consider a hash join of table A and B, where table B is the inner table. If the column value of NUM_DISTINCT data from the DBA_TAB_COLUMN dictionary table is small, then this implies that most of the rows have the same column value.

For example, the table emp has a gender column with two distinct values: male and female. It is assumed that queries on the gender column have a selectivity of one divided by two, or 50%. This means that half of the table rows are fetched. In this particular case, a hash join is most efficient.

> **See Also:** For more information, see "Verifying Column Statistics" in Chapter 8, "Gathering Statistics".

Note: The optimizer can use either a full compute (e.g., full table scan) or it can estimate by using a sample of the data. The problem with estimation based on data sampling is that the sample rows selected from data blocks could all be skewed. If there are 8 million rows in the table, then the optimizer may only consider a random subset and generate statistics based on that subset.

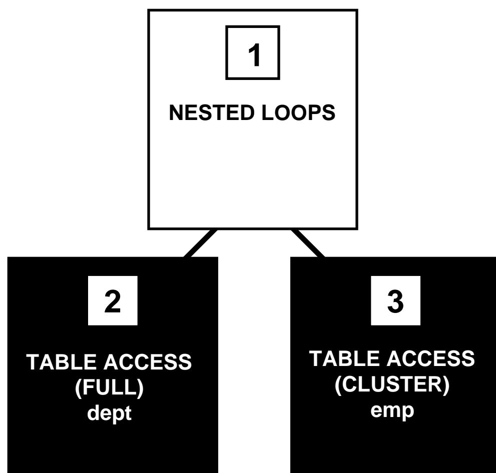
Cluster Join

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.

Figure 4–7 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 4–7 Cluster Join



To execute this statement, Oracle performs the following 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. Because 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.

How the Optimizer Chooses the Join Method

The optimizer costs each join method and chooses the method with the least cost. If a join returns many rows, then the optimizer considers the following three factors:

- A nested loops join (NL) is inefficient when a join returns a large number of rows [typically, more than 10,000 rows is considered large], and the optimizer may choose not to use it.

  The cost of a nested loops join = access cost of A + (access cost of B * number of rows from A)

- If you are using the RBO, then a merge join is the most efficient join when a join returns a large number or rows.

The cost of a merge join = access cost of A + access cost of B + (sort cost of A + sort cost of B)

An exception is when the data is pre-sorted. In the pre-sorted case, merge join costs = access cost of A + access cost of B where (sort cost of A + sort cost of B) = 0.

- If you are using the CBO, then a hash join is the most efficient join when a join returns a large number or rows.

  Estimated costs to perform a hash join = (access cost of A * number of hash partitions of B) + access cost of B

Forcing the Join Order

The following example illustrates the use of the `ORDERED` hint, which specifies the join order that the optimizer should use when joining tables. The `ORDERED` hint causes the join order to proceed in the order that the tables are listed in the `FROM` clause. In this example, the optimizer will start with the table `jl_br_journals` first, followed by `jl_br_balances`, followed by `gl_code_combinations`, etc. When using the `ORDERED` hint, it is important that the tables in the `FROM` clause are listed in the correct order, so as to prevent Cartesian joins.

```
SELECT /*+ ORDERED */
      glcc.segment1||' '||glcc.segment2||' '||glcc.segment3||' '
        ||glcc.segment4||' ' ||glcc.segment5 account,
 glcc.code_combination_id ccid, REPLACE(SUBSTR(glf.description,1,40),'.',' '),
 b.application_id, b.set_of_books_id, b.personnel_id, p.vendor_id
FROM  jl_br_journals j,
      jl_br_balances b,
      gl_code_combinations glcc,
      fnd_flex_values_vl glf,
      gl_periods gp,
      gl_sets_of_books gsb,
      po_vendors p
WHERE  j.application_id = b.application_id(+) AND
       j.set_of_books_id = b.set_of_books_id(+) AND
j.code_combination_id = b.code_combination_id(+) AND
       j.personnel_id = b.personnel_id(+) AND j.period_name = b.period_name(+) AND
       j.code_combination_id= glcc.code_combination_id AND j.period_name = gp.period_name AND
       j.set_of_books_id = gsb.set_of_books_id AND gp.period_set_name = gsb.period_set_name AND
       glcc.segment1 || '' = '01' AND glf.flex_value_set_id||'' = :c_account_vs AND
       glcc.segment3 = glf.flex_value AND gp.start_date = add_months('01-SEP-98',-1) AND
       gp.period_set_name = gsb.period_set_name AND j.application_id = 200 AND
       j.set_of_books_id = 225 AND j.personnel_id = p.vendor_id
GROUP BY glcc.segment1||' '||glcc.segment2||' '||glcc.segment3||
         ' '||glcc.segment4||' '||glcc.segment5,
```

```
        glcc.code_combination_id, REPLACE(SUBSTR(glf.description,1,40),'.',' '),
        b.application_id, b.set_of_books_id, b.personnel_id, p.vendor_id

Cost=13 SELECT STATEMENT
Cost=13   SORT GROUP BY
Cost=11     NESTED LOOPS
Cost=10       NESTED LOOPS
Cost=9          NESTED LOOPS
Cost=7            NESTED LOOPS
Cost=6              NESTED LOOPS
Cost=3                NESTED LOOPS
Cost=2                  NESTED LOOPS OUTER
Cost=1                    TABLE ACCESS BY INDEX ROWID JL_BR_JOURNALS_ALL
Cost=2                      INDEX RANGE SCAN JL_BR_JOURNALS_U1:
Cost=1                    TABLE ACCESS FULL JL_BR_BALANCES_ALL
Cost=1                  TABLE ACCESS BY INDEX ROWID GL_CODE_COMBINATIONS
Cost=                     INDEX UNIQUE SCAN GL_CODE_COMBINATIONS_U1:
Cost=3                TABLE ACCESS BY INDEX ROWID FND_FLEX_VALUES
Cost=2                  INDEX RANGE SCAN FND_FLEX_VALUES_N1:
Cost=1              TABLE ACCESS BY INDEX ROWID FND_FLEX_VALUES_TL
Cost=                 INDEX UNIQUE SCAN FND_FLEX_VALUES_TL_U1:
Cost=2            TABLE ACCESS BY INDEX ROWID GL_PERIODS
Cost=1              INDEX RANGE SCAN GL_PERIODS_N1:
Cost=1          TABLE ACCESS BY INDEX ROWID GL_SETS_OF_BOOKS
Cost=             INDEX UNIQUE SCAN GL_SETS_OF_BOOKS_U2:
Cost=1        TABLE ACCESS BY INDEX ROWID PO_VENDORS
Cost=           INDEX UNIQUE SCAN PO_VENDORS_U1:
```

Choosing Execution Plans for Join Statements

This section describes how the optimizer chooses an execution plan for a join statement:

- Choosing Execution Plans for Joins with the CBO
- Choosing Execution Plans for Joins with the RBO

The following considerations apply to both 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, then 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 CBO

With the CBO, 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, then 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 CBO, 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, then the optimizer ignores the hint and chooses the order. You can also override the optimizer's choice of join operations with hints.

> **See Also:** For more information on using hints, see Chapter 7, "Using Optimizer Hints".

Choosing Execution Plans for Joins with the RBO

With the rule-based approach, the optimizer performs the following 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 described in Chapter 4, "The Optimizer". 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, then 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 if there is an equijoin condition between the chosen table and the previous table or row source in join order, then the optimizer chooses a sort-merge operation.

   – If the access path for the chosen table is ranked lower than 12, and if there is not an equijoin condition, then 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. Because 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, then the optimizer chooses the execution plan with the fewest sort-merge operations.

c. If there is still a tie, then 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, then 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, then 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, then the optimizer chooses the execution plan for which the first table appears later in the query's `FROM` clause.

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`, then 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.

> **See Also:** For information about optimizer hints, see Chapter 7, "Using 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 CBO recognizes star queries and generates efficient execution plans for them. (Star queries are not recognized by the RBO.)

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 or a separate bitmap index on each key column.

> **See Also:** For more information about tuning star queries, see *Oracle8i Data Warehousing Guide*.

Optimizing Statements that Use Common Subexpressions

Common subexpression elimination is an optimization heuristic that identifies, removes, and collects common subexpression from disjunctive (i.e., OR) branches of a query. In most cases, it results in the reduction of the number of joins that would be performed.

Common subexpression elimination is enabled with initialization parameter OPTIMIZER_FEATURES_ENABLE or by setting the _ELIMINATE_COMMON_SUBEXPR parameter to TRUE.

A query is considered valid for common sub-expression elimination if its WHERE clause is in following form:

1. The top-level must be a disjunction; that is, a list of ORed logs.
2. Each disjunct must be either a simple predicate or a conjunction; that is, a list of ANDed logs.
3. Each conjunct must be either a simple predicate or a disjunction of simple predicates. (A predicate is considered simple if it does not contain AND or OR.)
4. An expression is considered common if it appears in all the disjunctive branches of the query.

Examples of Common Subexpression Elimination

The following query finds names of employees who work in a department located in L.A. *and* who make more than 40K *or* who are accountants.

```
SELECT emp.ename
FROM emp E, dept D
WHERE (D.deptno = E.deptno AND E.position = 'Accountant' AND D.location ='L.A.')
    OR
    E.deptno = D.deptno AND E.sal > 40000 AND D.location = 'L.A.');
```

The following query contains common subexpressions in its two disjunctive branches. The elimination of the common subexpressions transforms this query into the following query, thereby reducing the number of joins from two to one.

```
SELECT emp.ename FROM emp E, dept D
WHERE (D.deptno = E.deptno AND D.location = 'L.A.')
    AND (E.position = 'Accountant' OR E.sal > 40000);
```

The following query contains common subexpression in its three disjunctive branches:

```
SELECT SUM (l_extendedprice* (1 - l_discount))
FROM PARTS, LINEITEM
WHERE (p_partkey = l_partkey
        AND p_brand = 'Brand#12'
        AND p_container IN ('SM CASE', 'SM BOX', 'SM PACK', 'SM PKG')
        AND l_quantity >= 1 AND l_quantity <= 1 + 10
        AND p_size >= 1 AND p_size <= 5
        AND l_shipmode IN ('AIR', 'REG AIR')
        AND l_shipinstruct = 'DELIVER IN PERSON')
    OR (l_partkey = p_partkey)
        AND p_brand = 'Brand#23'
        AND p_container IN ('MED BAG', 'MED BOX', 'MED PKG', 'MED PACK')
        AND l_quantity >= 10 AND l_quantity <= 10 + 10
        AND p_size >= 1 AND p_size <= 10 AND p_size BETWEEN 1 AND 10
        AND l_shipmode IN ('AIR', 'REG AIR')
        AND l_shipinstruct = 'DELIVER IN PERSON')
    OR (p_partkey = l_partkey
        AND p_brand = 'Brand#34'
        AND p_container IN ('LG CASE', 'LG BOX', 'LG PACK', 'LG PKG')
        AND l_quantity >= 20 AND l_quantity <= 20 + 10
        AND p_size >= 1 AND p_size <= 15
        AND l_shipmode IN ('AIR', 'REG AIR')
        AND l_shipinstruct = 'DELIVER IN PERSON');
```

The above query is transformed by common subexpression elimination as the following, thereby reducing the number joins from three down to one.

```
SELECT SUM (l_extendedprice* (1 - l_discount))
FROM PARTS, LINEITEM
WHERE (p_partkey = l_partkey  /* these are the four common subexpressions */
    AND p_size >= 1
    AND l_shipmode IN ('AIR', 'REG AIR')
    AND l_shipinstruct = 'DELIVER IN PERSON')
    AND
        ((p_brand = 'Brand#12'
        AND p_container IN  ( 'SM CASE', 'SM BOX', 'SM PACK', 'SM PKG')
        AND l_quantity >= 1 AND l_quantity <= 1 + 10
        AND p_size <= 5)
    OR (p_brand = 'Brand#23'
        AND p_container IN ('MED BAG', 'MED BOX', 'MED PKG', 'MED PACK')
        AND l_quantity >= 10 AND l_quantity <= 10 + 10
        AND p_size <= 10)
    OR (p_brand = 'Brand#34'
        AND p_container IN ('LG CASE', 'LG BOX', 'LG PACK', 'LG PKG')
        AND l_quantity >= 20 AND l_quantity <= 20 + 10
```

```
AND p_size <= 15));
```

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.

This section discusses how the optimizer evaluates expressions and conditions that contain the following:

- Constants
- LIKE Operator
- IN Operator
- ANY or SOME Operator
- ALL Operator
- BETWEEN Operator
- NOT Operator
- Transitivity
- DETERMINISTIC Functions

Constants

Computation of constants is performed only once, when the statement is optimized, rather than each time the statement is executed.

For example, the following conditions test for monthly salaries greater than 2000:

```
sal > 24000/12

sal > 2000

sal*12 > 24000
```

If a SQL statement contains the first condition, then the optimizer simplifies it into the second condition.

Note: 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), then 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 Also: For more information, see "Example 2: IN Subquery" on page 4-79.

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 then 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, then 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 CBO.

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	Any of the comparison operators =, !=, ^=, <, <>, >, <=, or >=.
constant	Any constant expression involving operators, SQL functions, literals, bind variables, and correlation variables.

Example: In the following query, the `WHERE` clause contains two conditions, each of 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, then 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
```

DETERMINISTIC Functions

In some cases, the optimizer can use a previously calculated value, rather than executing a user-written function. This is only safe for functions that behave in a restricted manner. The function must always return the same output return value for any given set of input argument values.

The function's result must not differ because of differences in the content of package variables or the database, or session parameters such as the NLS parameters. Furthermore, if the function is redefined in the future, then its output return value must still be the same as that calculated with the prior definition for any given set of input argument values. Finally, there must be no meaningful side-effects such that using a precalculated value instead of executing the function again would alter the application.

The creator of a function can promise to the Oracle server that the function behaves according to these restrictions by using the keyword `DETERMINISTIC` when declaring the function with a `CREATE FUNCTION` statement or in a `CREATE PACKAGE` or `CREATE TYPE` statement. The server does not attempt to verify this declaration—even a function that obviously manipulates the database or package variables can be declared `DETERMINISTIC`. It is the programmer's responsibility to use this keyword only when appropriate.

Calls to a `DETERMINISTIC` function may be replaced by the use of an already calculated value when the function is called multiple times within the same query, or if there is a function-based index or a materialized view defined that includes a relevant call to the function.

See Also:

- For more information on DETERMINISTIC functions, see *Oracle8i Application Developer's Guide - Fundamentals*.
- For descriptions of CREATE FUNCTION, CREATE INDEX, and CREATE MATERIALIZED VIEW, see *Oracle8i SQL Reference*.
- For a description of function-based indexes, see *Oracle8i Concepts*.
- For detailed information about materialized views, see *Oracle8i Data Warehousing Guide*.

Transforming and Optimizing Statements

SQL is a very flexible query language; there are often many statements you could use 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

See Also: For additional information about optimizing statements that contain joins, semi-joins, or anti-joins, see "Optimizing Joins" on page 4-49.

Transforming ORs into Compound Queries

If a query contains a WHERE clause with multiple conditions combined with OR operators, then 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, then 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, then 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 CBO, 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 CBO does not use the OR transformation for IN-lists or ORs on the same column; instead, it uses the INLIST iterator operator.

> **See Also:** For information on access paths and how indexes make them available, see the "Access Paths for the RBO" section on page 4-34 and "How the CBO Chooses an Access Path" on page 4-25.

Example: In the following query, the WHERE clause 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, then the optimizer may transform this query into the equivalent query below:

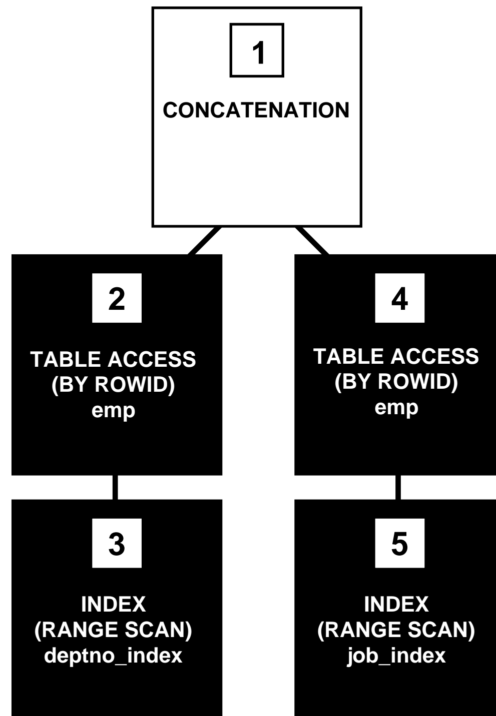
```
SELECT *
  FROM emp
  WHERE job = 'CLERK'
UNION ALL
SELECT *
  FROM emp
  WHERE deptno = 10
    AND job <> 'CLERK';
```

When the CBO is deciding whether to make a transformation, the optimizer compares the cost of executing the original query using a full table scan with that of executing the resulting query.

With the RBO, the optimizer makes this UNION ALL transformation, because each component query of the resulting compound query can be executed using an index. The RBO 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 4–8.

Figure 4–8 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, then 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: The following query assumes 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;
```

Because 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 the following:

- Transform the complex statement into an equivalent join statement, and then optimize the join statement.
- Optimize the complex statement as it 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 optimizer techniques described in "Optimizing Joins" on page 4-49.

The following complex statement selects all rows from the `accounts` table whose owners appear in the `customers` table:

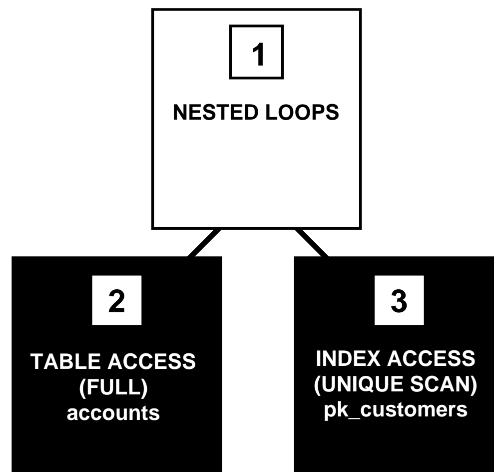
```
SELECT *
  FROM accounts
  WHERE custno IN
    (SELECT custno FROM customers);
```

If the `custno` column of the `customers` table is a primary key or has a `UNIQUE` constraint, then the optimizer can transform the complex query into the following 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 4–9.

Figure 4–9 Execution Plan for a Nested Loops Join



To execute this statement, Oracle performs a nested-loops join operation.

> **See Also:** For information on nested loops joins, see "Optimizing Joins" on page 4-49.

If the optimizer cannot transform a complex statement into a join statement, then 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.

The following complex statement 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:** Complex queries whose subqueries contain aggregate 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 the following:

- 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 the following 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 non-mergeable 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 4-88.)

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, aggregate functions, `DISTINCT`, `GROUP BY`, `CONNECT BY`, and so on (as described in "Mergeable and Non-mergeable Views" on page 4-77).

Example: The following view is 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;
```

The following query 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, then the resulting `WHERE` clause makes them available.

Mergeable and Non-mergeable 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 the following:

- Set operators (`UNION`, `UNION ALL`, `INTERSECT`, `MINUS`)
- A `CONNECT BY` clause
- A `ROWNUM` pseudocolumn
- Aggregate 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* (described below) is enabled:

- 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. However, if a view on the right side of an outer join has only one base table, then the optimizer can use complex view merging, even if an expression in the view can return a non-null value for a `NULL`.

> **See Also:** For more information, see "Optimizing Joins" on page 4-49.

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 4-79).

Complex merging is not cost-based—it must be enabled with the initialization parameter OPTIMIZER_FEATURES_ENABLE, the MERGE hint, or the parameter _COMPLEX_VIEW_MERGING. Without this hint or parameter setting, the optimizer uses another approach (see "Pushing the Predicate into the View" on page 4-79).

> **See Also:** For details about the MERGE and NO_MERGE hints, see Chapter 7, "Using Optimizer Hints".

Example 1: View with a GROUP BY Clause The view avg_salary_view 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 the following query, which finds the average salaries of departments in London:

```
SELECT dept.loc, avg_sal_dept
  FROM dept, avg_salary_view
  WHERE dept.deptno = avg_salary_view.deptno
    AND dept.loc = 'London';
```

into the following query:

```
SELECT dept.loc, AVG(sal)
  FROM dept, emp
  WHERE dept.deptno = emp.deptno
    AND dept.loc = 'London'
  GROUP BY dept.rowid, dept.loc;
```

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. The view `min_salary_view` 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 the following 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.loc = 'London';
```

into the following 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.loc = '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 a non-mergeable view by pushing the query block's predicates inside the view's query.

Example 1: The `two_emp_tables` view 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;
```

The following query 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.

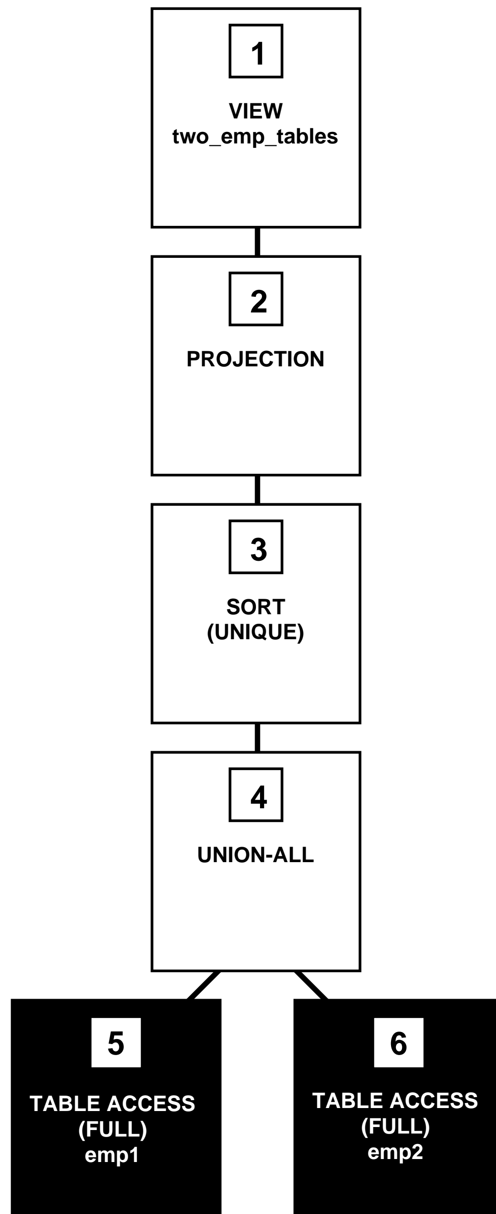
The resulting statement looks like the following:

```
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, then the resulting `WHERE` clauses make it available.

Figure 4–10 shows the execution plan of the resulting statement.

Figure 4–10 Accessing a View Defined with the UNION Set Operator



To execute this statement, Oracle performs the following 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: The view `emp_group_by_deptno` 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;
```

The following query selects the average, minimum, and maximum salaries of department 10 from the `emp_group_by_deptno` view:

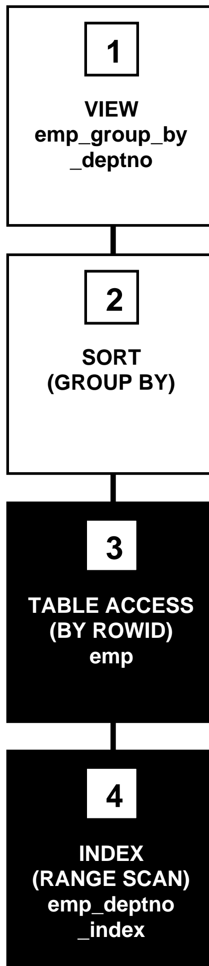
```
SELECT *
  FROM emp_group_by_deptno
  WHERE deptno = 10;
```

The optimizer transforms the statement by pushing its predicate (the `WHERE` clause condition) into the view's query. The resulting statement looks like the following:

```
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, then the resulting `WHERE` clause makes it available. Figure 4–11 shows the execution plan for the resulting statement. The execution plan uses an index on the `deptno` column.

Figure 4–11 Accessing a View Defined with a GROUP BY Clause



To execute this statement, Oracle performs the following 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 an Aggregate Function to the View The optimizer can transform a query that contains an aggregate function (`AVG`, `COUNT`, `MAX`, `MIN`, `SUM`) by applying the function to the view's query.

Example: The following query 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:

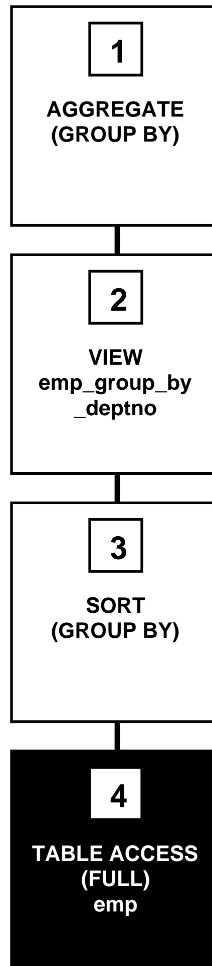
```
SELECT AVG(avg_sal), AVG(min_sal), AVG(max_sal)
    FROM emp_group_by_deptno;
```

The optimizer transforms this statement by applying the `AVG` aggregate 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 4–12 shows the execution plan of the resulting statement.

Figure 4–12 Applying Aggregate 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.

Views in Outer Joins

For a view that is on the right side of an outer join, the optimizer can use one of two methods, depending on how many base tables the view accesses:

- If the view has only one base table, then the optimizer can use *view merging*.
- If the view has multiple base tables, then 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 4-76.) 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.loc = '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.loc = '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 4-79) if the initialization parameter _PUSH_JOIN_PREDICATE is set to TRUE or the accessing query contains the PUSH_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 loops joins, and full table scans to index scans.

> **See Also:** For information about optimizer hints, see Chapter 7, "Using Optimizer Hints".

Example 1: 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.loc = '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.loc = 'London')
  WHERE bonus.job IN ('engineer', 'accountant');
```

Example 2: Consider the following example:

```
SELECT 'PAYMENT' c_tx_type, c.check_id c_tx_id, 1 c_je_header_id,
    c.status_lookup_code, c_tx_status, DECODE(:c_bank_curr_dsp,:c_gl_currency_
    code, NVL(c.base_amount,NVL(c.amount,0)), NVL(c.amount,0)) c_tx_ba_amount,
    DECODE(SIGN(:c_julian_as_of_date -
    TO_CHAR(c.check_date,'J')),-1, DECODE(:c_bank_curr_dsp,:c_gl_currency_code,
    NVL(c.base_amount,NVL(c.amount,0)), NVL(c.amount,0)),0) c_tx_ba_future_
    amount, NULL c_tx_dr_cr, cs.future_pay_code_combination_id c_tx_clearing_
    ccid,  NVL(c.exchange_rate, 0) c_tx_exchange_rate
FROM  ap_checks c,
      ap_check_stocks cs
WHERE (c.check_stock_id(+) = cs.check_stock_id ) AND
      (:c_sl_reference_type = 'PAYMENT') AND
      (:c_sl_reference_id= c.check_id) AND (:c_sl_je_header_id = 1);
```

Without pushing the join predicate: 41 minutes, 1,492,141 buffer gets, 125,202 disk reads

```
Cost=20003 SELECT STATEMENT
Cost=   FILTER
Cost=     FILTER
Cost=       NESTED LOOPS OUTER
Cost=1         TABLE ACCESS FULL AP_CHECK_STOCKS_ALL
Cost=20002         TABLE ACCESS FULL AP_CHECKS_ALL
```

After pushing the join predicate: 0.01 seconds, 6 buffer gets, 5 disk reads

```
Cost=4 SELECT STATEMENT
Cost=   FILTER
Cost=4     NESTED LOOPS OUTER
Cost=3       TABLE ACCESS BY INDEX ROWID AP_CHECKS_ALL
Cost=2         INDEX UNIQUE SCAN AP_CHECKS_U1:
Cost=1       TABLE ACCESS BY INDEX ROWID AP_CHECK_STOCKS_ALL
Cost=         INDEX UNIQUE SCAN AP_CHECK_STOCKS_U1:
```

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, then 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;
```

The following query 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:

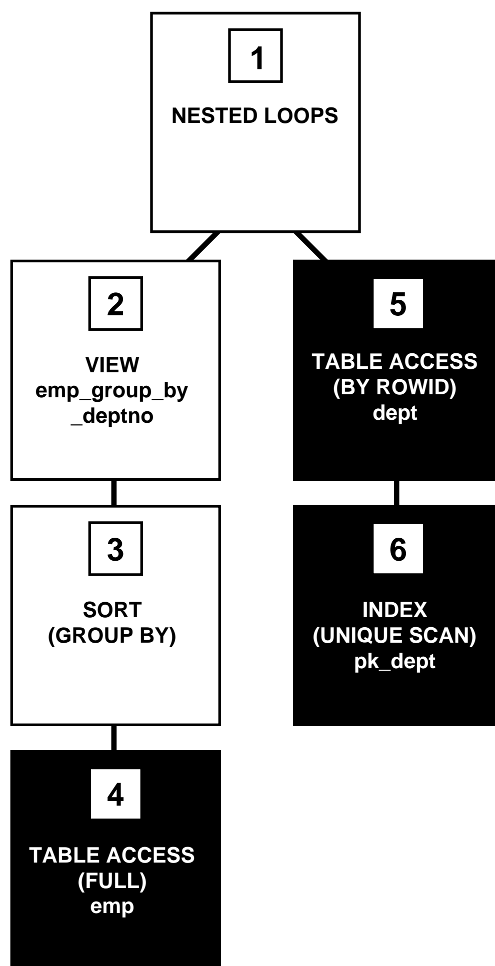
```
SELECT emp_group_by_deptno.deptno, avg_sal, min_sal,
  max_sal, dname, loc
    FROM emp_group_by_deptno, dept
    WHERE emp_group_by_deptno.deptno = dept.deptno;
```

Because 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.

> **See Also:** For more information on how Oracle performs a nested loops join operation, see "Optimizing Joins" on page 4-49.

Figure 4–13 shows the execution plan for this statement.

Figure 4–13 Joining a View Defined with a GROUP BY Clause to a Table



To execute this statement, Oracle performs the following 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.

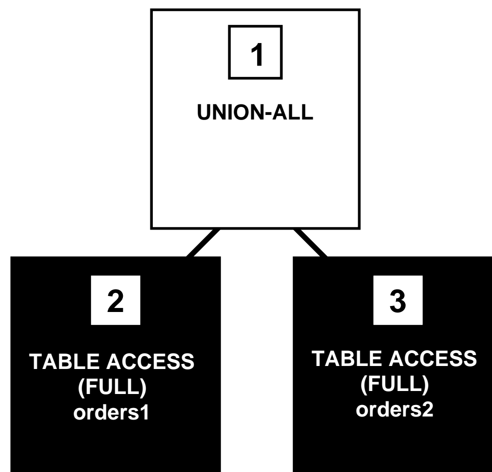
Optimizing Compound Queries

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 4–14 shows the execution plan for the following 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 4–14 Compound Query with UNION ALL Set Operator



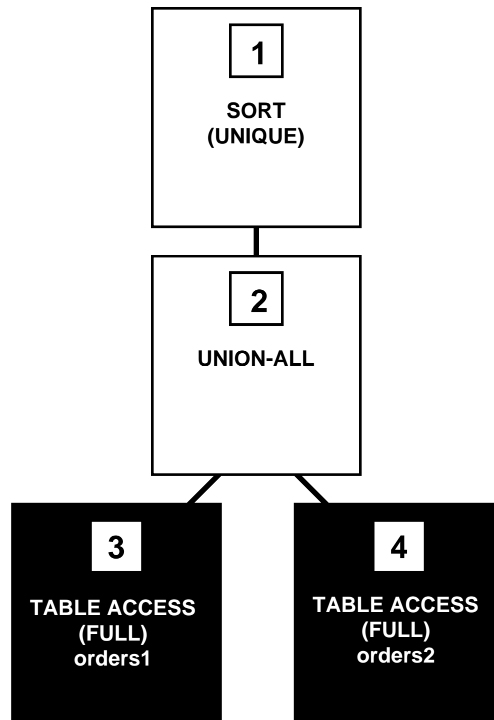
To execute this statement, Oracle performs the following steps:

- 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 4–15 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 4–15 Compound Query with UNION Set Operator

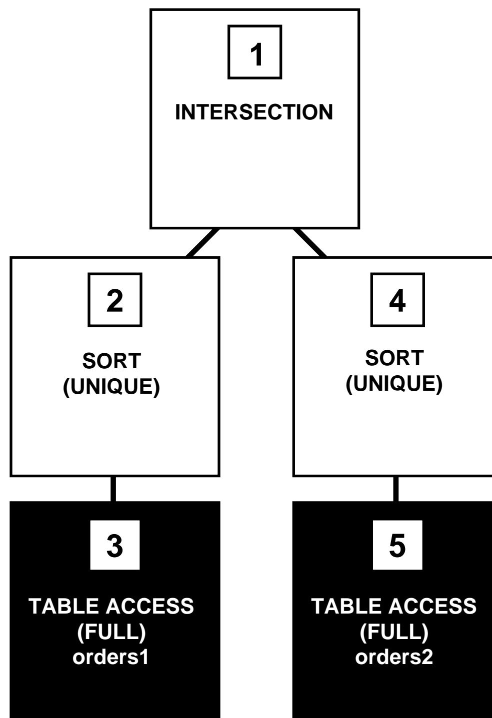


This execution plan is identical to the one for the `UNION-ALL` operator shown in Figure 4–14 on page 4-91, except that in this case, Oracle uses the `SORT` operation to eliminate the duplicates returned by the `UNION-ALL` operation.

Figure 4–16 shows the execution plan for the following 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 4–16 Compound Query with INTERSECT Set Operator



To execute this statement, Oracle performs the following 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 that 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, then 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, then 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 the CBO. 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.

> **See Also:** For more information on tuning distributed queries, see Chapter 9, "Optimizing SQL Statements".

5

Using EXPLAIN PLAN

This chapter introduces execution plans, describes the SQL statement `EXPLAIN PLAN`, and explains how to interpret its output. This chapter also discusses plan stability features and the use of stored outlines to preserve your tuning investment for particular SQL statements. This chapter provides procedures for managing outlines to control application performance characteristics.

This chapter contains the following sections:

- Understanding EXPLAIN PLAN
- Creating the Output Table
- Displaying PLAN_TABLE Output
- Output Table Columns
- Bitmap Indexes and EXPLAIN PLAN
- EXPLAIN PLAN and Partitioned Objects
- EXPLAIN PLAN Restrictions

> **See Also:** For the syntax of `EXPLAIN PLAN`, see the *Oracle8i SQL Reference.*

Understanding EXPLAIN PLAN

The EXPLAIN PLAN statement displays execution plans chosen by the Oracle optimizer for SELECT, UPDATE, INSERT, and DELETE statements. A statement's execution plan is the sequence of operations Oracle performs to execute the statement. The components of execution plans include:

- An ordering of the tables referenced by the statement.
- An access method for each table mentioned in the statement.
- A join method for tables affected by join operations in the statement.

EXPLAIN PLAN output shows how Oracle executes SQL statements. EXPLAIN PLAN results alone, however, cannot differentiate between well-tuned statements and those that perform poorly. For example, if EXPLAIN PLAN output shows that a statement uses an index, then this does not mean the statement runs efficiently. Sometimes using indexes can be extremely inefficient. It is best to use EXPLAIN PLAN to determine an access plan, and later prove that it is the optimal plan through testing.

When evaluating a plan, always examine the statement's actual resource consumption. For best results, use the Oracle Trace or SQL trace facility and TKPROF to examine individual SQL statement performance.

See Also: Chapter 6, "Using SQL Trace and TKPROF" and Chapter 14, "Using Oracle Trace".

Creating the Output Table

Before issuing an EXPLAIN PLAN statement, create a table to hold its output. Use one of the following approaches:

- Run the SQL script UTLXPLAN.SQL to create a sample output table called PLAN_TABLE in your schema. The exact name and location of this script depends on your operating system. For example, on Sun Solaris, the UTLXPLAN.SQL is located under $ORACLE_HOME/rdbms/admin. PLAN_TABLE is the default table into which the EXPLAIN PLAN statement inserts rows describing execution plans.
- Issue a CREATE TABLE statement to create an output table with any name you choose. When you issue an EXPLAIN PLAN statement, you can direct its output to this table.

Any table used to store the output of the EXPLAIN PLAN statement must have the same column names and datatypes as the PLAN_TABLE:

```
CREATE TABLE PLAN_TABLE (
    STATEMENT_ID    VARCHAR2(30),
    TIMESTAMP       DATE,
    REMARKS         VARCHAR2(80),
    OPERATION       VARCHAR2(30),
    OPTIONS         VARCHAR2(30),
    OBJECT_NODE     VARCHAR2(128),
    OBJECT_OWNER    VARCHAR2(30),
    OBJECT_NAME     VARCHAR2(30),
    OBJECT_INSTANCE NUMERIC,
    OBJECT_TYPE     VARCHAR2(30),
    OPTIMIZER       VARCHAR2(255),
    SEARCH_COLUMNS  NUMBER,
    ID              NUMERIC,
    PARENT_ID       NUMERIC,
    POSITION        NUMERIC,
    COST            NUMERIC,
    CARDINALITY     NUMERIC,
    BYTES           NUMERIC,
    OTHER_TAG       VARCHAR2(255),
    PARTITION_START VARCHAR2(255),
    PARTITION_STOP  VARCHAR2(255),
    PARTITION_ID    NUMERIC,
    OTHER           LONG,
    DISTRIBUTION    VARCHAR2(30));
```

Displaying PLAN_TABLE Output

Display the most recent plan table output using the following scripts:

- UTLXPLS.SQL - Shows plan table output for serial processing.
- UTLXPLP.SQL - Shows plan table output with parallel execution columns.

The row source count values in EXPLAIN PLAN output identify the number of rows processed by each step in the plan. This helps you identify inefficiencies in the query; for example, the row source with an access plan that is performing inefficient operations.

Output Table Columns

The PLAN_TABLE used by the EXPLAIN PLAN statement contains the following columns:

Table 5–1 PLAN_TABLE Columns **(Page 1 of 3)**

Column	Description
STATEMENT_ID	The value of the optional STATEMENT_ID parameter specified in the EXPLAIN PLAN statement.
TIMESTAMP	The date and time when the EXPLAIN PLAN statement was issued.
REMARKS	Any comment (of up to 80 bytes) you want to associate with each step of the explained plan. If you need to add or change a remark on any row of the PLAN_TABLE, then use the UPDATE statement to modify the rows of the PLAN_TABLE.
OPERATION	The name of the internal operation performed in this step. In the first row generated for a statement, the column contains one of the following values: DELETE STATEMENT INSERT STATEMENT SELECT STATEMENT UPDATE STATEMENT See Table 5–4 for more information on values for this column.
OPTIONS	A variation on the operation described in the OPERATION column. See Table 5–4 for more information on values for this column.
OBJECT_NODE	The name of the database link used to reference the object (a table name or view name). For local queries using parallel execution, this column describes the order in which output from operations is consumed.

Table 5–1 PLAN_TABLE Columns **(Page 2 of 3)**

OBJECT_OWNER	The name of the user who owns the schema containing the table or index.
OBJECT_NAME	The name of the table or index.
OBJECT_INSTANCE	A number corresponding to the ordinal position of the object as it appears in the original statement. The numbering proceeds from left to right, outer to inner with respect to the original statement text. View expansion results in unpredictable numbers.
OBJECT_TYPE	A modifier that provides descriptive information about the object; for example, NON-UNIQUE for indexes.
OPTIMIZER	The current mode of the optimizer.
SEARCH_COLUMNS	Not currently used.
ID	A number assigned to each step in the execution plan.
PARENT_ID	The ID of the next execution step that operates on the output of the ID step.
POSITION	The order of processing for steps that all have the same PARENT_ID.
COST	The cost of the operation as estimated by the optimizer's cost-based approach. For statements that use the rule-based approach, this column is null. Cost is not determined for table access operations. The value of this column does not have any particular unit of measurement, it is merely a weighted value used to compare costs of execution plans.
CARDINALITY	The estimate by the cost-based approach of the number of rows accessed by the operation.
BYTES	The estimate by the cost-based approach of the number of bytes accessed by the operation.
OTHER_TAG	Describes the contents of the OTHER column. See Table 5–2 for more information on the possible values for this column.

Table 5–1 PLAN_TABLE Columns (Page 3 of 3)

PARTITION_START	The start partition of a range of accessed partitions. It can take one of the following values: *n* indicates that the start partition has been identified by the SQL compiler, and its partition number is given by *n*. KEY indicates that the start partition will be identified at execution time from partitioning key values. ROW LOCATION indicates that the start partition (same as the stop partition) will be computed at execution time from the location of each record being retrieved. The record location is obtained by a user or from a global index. INVALID indicates that the range of accessed partitions is empty.
PARTITION_STOP	The stop partition of a range of accessed partitions. It can take one of the following values: *n* indicates that the stop partition has been identified by the SQL compiler, and its partition number is given by *n*. KEY indicates that the stop partition will be identified at execution time from partitioning key values. ROW LOCATION indicates that the stop partition (same as the start partition) will be computed at execution time from the location of each record being retrieved. The record location is obtained by a user or from a global index. INVALID indicates that the range of accessed partitions is empty.
PARTITION_ID	The step that has computed the pair of values of the PARTITION_START and PARTITION_STOP columns.
OTHER	Other information that is specific to the execution step that a user may find useful.
DISTRIBUTION	Stores the method used to distribute rows from *producer* query servers to *consumer* query servers. See Table 5–3 for more information on the possible values for this column. For more information about consumer and producer query servers, see *Oracle8i Concepts*.

Table 5–2 describes the values that may appear in the OTHER_TAG column.

Table 5–2 Values of OTHER_TAG Column of the PLAN_TABLE

OTHER_TAG Text (examples)	Meaning	Interpretation
blank		Serial execution.
SERIAL_FROM_REMOTE (S -> R)	Serial from remote	Serial execution at a remote site.
SERIAL_TO_PARALLEL (S -> P)	Serial to parallel	Serial execution; output of step is partitioned or broadcast to parallel execution servers.
PARALLEL_TO_PARALLEL (P - > P)	Parallel to parallel	Parallel execution; output of step is repartitioned to second set of parallel execution servers.
PARALLEL_TO_SERIAL (P -> S)	Parallel to serial	Parallel execution; output of step is returned to serial "query coordinator" process.
PARALLEL_COMBINED_ WITH_PARENT (PWP)	Parallel combined with parent	Parallel execution; output of step goes to next step in same parallel process. No interprocess communication to parent.
PARALLEL_COMBINED_ WITH_CHILD (PWC)	Parallel combined with child	Parallel execution; input of step comes from prior step in same parallel process. No interprocess communication from child.

Table 5–3 describes the values that can appear in the DISTRIBUTION column:

Table 5–3 Values of DISTRIBUTION Column of the PLAN_TABLE

DISTRIBUTION Text	Interpretation
PARTITION (ROWID)	Maps rows to query servers based on the partitioning of a table or index using the rowid of the row to UPDATE/DELETE.
PARTITION (KEY)	Maps rows to query servers based on the partitioning of a table or index using a set of columns. Used for partial partition-wise join, PARALLEL INSERT, CREATE TABLE AS SELECT of a partitioned table, and CREATE PARTITIONED GLOBAL INDEX.
HASH	Maps rows to query servers using a hash function on the join key. Used for PARALLEL JOIN or PARALLEL GROUP BY.
RANGE	Maps rows to query servers using ranges of the sort key. Used when the statement contains an ORDER BY clause.
ROUND-ROBIN	Randomly maps rows to query servers.
BROADCAST	Broadcasts the rows of the entire table to each query server. Used for a parallel join when one table is very small compared to the other.
QC (ORDER)	The query coordinator consumes the input in order, from the first to the last query server. Used when the statement contains an ORDER BY clause.
QC (RANDOM)	The query coordinator consumes the input randomly. Used when the statement does not have an ORDER BY clause.

Table 5–4 lists each combination of OPERATION and OPTION produced by the EXPLAIN PLAN statement and its meaning within an execution plan.

Table 5–4 OPERATION and OPTION Values Produced by EXPLAIN PLAN (Page 1 of 4)

Operation	Option	Description
AND-EQUAL		Operation accepting multiple sets of rowids, returning the intersection of the sets, eliminating duplicates. Used for the single-column indexes access path.
	CONVERSION	TO ROWIDS converts bitmap representations to actual rowids that can be used to access the table. FROM ROWIDS converts the rowids to a bitmap representation. COUNT returns the number of rowids if the actual values are not needed.
	INDEX	SINGLE VALUE looks up the bitmap for a single key value in the index. RANGE SCAN retrieves bitmaps for a key value range. FULL SCAN performs a full scan of a bitmap index if there is no start or stop key.
	MERGE	Merges several bitmaps resulting from a range scan into one bitmap.
	MINUS	Subtracts bits of one bitmap from another. Row source is used for negated predicates. Can be used only if there are nonnegated predicates yielding a bitmap from which the subtraction can take place. An example appears in "Bitmap Indexes and EXPLAIN PLAN" on page 5-13.
	OR	Computes the bitwise OR of two bitmaps.
CONNECT BY		Retrieves rows in hierarchical order for a query containing a CONNECT BY clause.
CONCATENATION		Operation accepting multiple sets of rows returning the union-all of the sets.
COUNT		Operation counting the number of rows selected from a table.
	STOPKEY	Count operation where the number of rows returned is limited by the ROWNUM expression in the WHERE clause.
DOMAIN INDEX		Retrieval of one or more rowids from a domain index.

Table 5–4 OPERATION and OPTION Values Produced by EXPLAIN PLAN (Page 2 of 4)

Operation	Option	Description
FILTER		Operation accepting a set of rows, eliminates some of them, and returns the rest.
FIRST ROW		Retrieval on only the first row selected by a query.
FOR UPDATE		Operation retrieving and locking the rows selected by a query containing a FOR UPDATE clause.
HASH JOIN		Operation joining two sets of rows and returning the result.
	ANTI	Hash anti-join.
(These are join operations.)	SEMI	Hash semi-join.
INDEX	UNIQUE SCAN	Retrieval of a single rowid from an index.
(These are access methods.)	RANGE SCAN	Retrieval of one or more rowids from an index. Indexed values are scanned in ascending order.
	RANGE SCAN DESCENDING	Retrieval of one or more rowids from an index. Indexed values are scanned in descending order.
INLIST ITERATOR		Iterates over the operation below it for each value in the IN-list predicate.
INTERSECTION		Operation accepting two sets of rows and returning the intersection of the sets, eliminating duplicates.
MERGE JOIN		Operation accepting two sets of rows, each sorted by a specific value, combining each row from one set with the matching rows from the other, and returning the result.
(These are join operations.)	OUTER	Merge join operation to perform an outer join statement.
	ANTI	Merge anti-join.
	SEMI	Merge semi-join.
CONNECT BY		Retrieval of rows in hierarchical order for a query containing a CONNECT BY clause.
MINUS		Operation accepting two sets of rows and returning rows appearing in the first set but not in the second, eliminating duplicates.

Table 5–4 OPERATION and OPTION Values Produced by EXPLAIN PLAN **(Page 3 of 4)**

Operation	Option	Description
NESTED LOOPS (These are join operations.)		Operation accepting two sets of rows, an outer set and an inner set. Oracle compares each row of the outer set with each row of the inner set, returning rows that satisfy a condition.
	OUTER	Nested loops operation to perform an outer join statement.
PARTITION	SINGLE	Access one partition.
	ITERATOR	Access many partitions (a subset).
	ALL	Access all partitions.
	INLIST	Similar to iterator, but based on an IN-list predicate.
	INVALID	Indicates that the partition set to be accessed is empty.
		Iterates over the operation below it, for each partition in the range given by the PARTITION_START and PARTITION_STOP columns. PARTITION describes partition boundaries applicable to a single partitioned object (table or index) or to a set of equi-partitioned objects (a partitioned table and its local indexes). The partition boundaries are provided by the values of PARTITION_START and PARTITION_STOP of the PARTITION. Refer to Table 5–1 for valid values of partition start/stop.
REMOTE		Retrieval of data from a remote database.
SEQUENCE		Operation involving accessing values of a sequence.
SORT	AGGREGATE	Retrieval of a single row that is the result of applying a group function to a group of selected rows.
	UNIQUE	Operation sorting a set of rows to eliminate duplicates.
	GROUP BY	Operation sorting a set of rows into groups for a query with a GROUP BY clause.
	JOIN	Operation sorting a set of rows before a merge-join.
	ORDER BY	Operation sorting a set of rows for a query with an ORDER BY clause.

Table 5–4 OPERATION and OPTION Values Produced by EXPLAIN PLAN (Page 4 of 4)

Operation	Option	Description
TABLE ACCESS	FULL	Retrieval of all rows from a table.
(These are access methods.)	CLUSTER	Retrieval of rows from a table based on a value of an indexed cluster key.
	HASH	Retrieval of rows from table based on hash cluster key value.
	BY ROWID	Retrieval of a row from a table based on its rowid.
	BY USER ROWID	If the table rows are located using user-supplied rowids.
	BY INDEX ROWID	If the table is nonpartitioned and rows are located using index(es).
	BY GLOBAL INDEX ROWID	If the table is partitioned and rows are located using only global indexes.
	BY LOCAL INDEX ROWID	If the table is partitioned and rows are located using one or more local indexes and possibly some global indexes.
		Partition Boundaries: The partition boundaries may have been computed by: A previous PARTITION step, in which case the PARTITION_START and PARTITION_STOP column values replicate the values present in the PARTITION step, and the PARTITION_ID contains the ID of the PARTITION step. Possible values for PARTITION_START and PARTITION_STOP are NUMBER(n), KEY, INVALID. The TABLE ACCESS or INDEX step itself, in which case the PARTITION_ID contains the ID of the step. Possible values for PARTITION_START and PARTITION_STOP are NUMBER(n), KEY, ROW LOCATION (TABLE ACCESS only), and INVALID.
UNION		Operation accepting two sets of rows and returns the union of the sets, eliminating duplicates.
VIEW		Operation performing a view’s query and then returning the resulting rows to another operation.

Note: Access methods and join operations are discussed in *Oracle8i Concepts.*

Bitmap Indexes and EXPLAIN PLAN

Index row sources using bitmap indexes appear in the EXPLAIN PLAN output with the word BITMAP indicating the type of the index. Consider the following sample query and plan:

```
EXPLAIN PLAN FOR
    SELECT * FROM t
    WHERE c1 = 2
    AND c2 <> 6
    OR c3 BETWEEN 10 AND 20;

SELECT STATEMENT
   TABLE ACCESS T BY INDEX ROWID
      BITMAP CONVERSION TO ROWID
         BITMAP OR
            BITMAP MINUS
               BITMAP MINUS
                  BITMAP INDEX  C1_IND  SINGLE VALUE
                  BITMAP INDEX  C2_IND  SINGLE VALUE
               BITMAP INDEX  C2_IND  SINGLE VALUE
            BITMAP MERGE
               BITMAP INDEX  C3_IND  RANGE SCAN
```

In this example, the predicate c1=2 yields a bitmap from which a subtraction can take place. From this bitmap, the bits in the bitmap for c2 = 6 are subtracted. Also, the bits in the bitmap for c2 IS NULL are subtracted, explaining why there are two MINUS row sources in the plan. The NULL subtraction is necessary for semantic correctness unless the column has a NOT NULL constraint. The TO ROWIDS option is used to generate the ROWIDs that are necessary for the table access.

EXPLAIN PLAN and Partitioned Objects

Use EXPLAIN PLAN to see how Oracle accesses partitioned objects for specific queries.

Partitions accessed after pruning are shown in the PARTITION START and PARTITION STOP columns. The row source name for the range partition is "PARTITION RANGE". For hash partitions, the row source name is PARTITION HASH.

A join is implemented using partial partition-wise join if the DISTRIBUTION column of the plan table of one of the joined tables contains PARTITION(KEY). Partial partition-wise join is possible if one of the joined tables is partitioned on its join column and the table is parallelized.

A join is implemented using full partition-wise join if the partition row source appears before the join row source in the EXPLAIN PLAN output. Full partition-wise joins are possible only if both joined tables are equi-partitioned on their respective join columns. Examples of execution plans for several types of partitioning follow.

Displaying Range and Hash Partitioning with EXPLAIN PLAN

Consider the following table, emp_range, partitioned by range on hiredate to illustrate how pruning is displayed. Assume that the tables emp and dept from a standard Oracle schema exist.

```
CREATE TABLE emp_range
PARTITION BY RANGE(hiredate)
(
    PARTITION emp_p1 VALUES LESS THAN (TO_DATE('1-JAN-1991','DD-MON-YYYY')),
    PARTITION emp_p2 VALUES LESS THAN (TO_DATE('1-JAN-1993','DD-MON-YYYY')),
    PARTITION emp_p3 VALUES LESS THAN (TO_DATE('1-JAN-1995','DD-MON-YYYY')),
    PARTITION emp_p4 VALUES LESS THAN (TO_DATE('1-JAN-1997','DD-MON-YYYY')),
    PARTITION emp_p5 VALUES LESS THAN (TO_DATE('1-JAN-1999','DD-MON-YYYY'))
)
AS SELECT * FROM emp;
```

Example 1a

```
EXPLAIN PLAN FOR SELECT * FROM emp_range;
```

Enter the following to display the EXPLAIN PLAN output:

```
@?/RDBMS/ADMIN/UTLXPLS
```

Oracle displays something similar to:

```
Plan Table
-------------------------------------------------------------------------------
| Operation                 |  Name     |  Rows | Bytes|  Cost  | Pstart |  Pstop|
-------------------------------------------------------------------------------
| SELECT STATEMENT          |           |   105 |    8K|      1 |        |       |
|  PARTITION RANGE ALL      |           |       |      |        |      1 |      5 |
|   TABLE ACCESS FULL       |EMP_RANGE  |   105 |    8K|      1 |      1 |      5 |
-------------------------------------------------------------------------------
6 rows selected.
```

A partition row source is created on top of the table access row source. It iterates over the set of partitions to be accessed.

In example 1a, the partition iterator covers all partitions (option `ALL`), because a predicate was not used for pruning. The `PARTITION_START` and `PARTITION_STOP` columns of the plan table show access to all partitions from 1 to 5.

Example 2a

```
EXPLAIN PLAN FOR SELECT * FROM emp_range
WHERE hiredate >= TO_DATE('1-JAN-1995','DD-MON-YYYY');

Plan Table
--------------------------------------------------------------------------------
| Operation                   | Name      |  Rows   | Bytes|  Cost   | Pstart| Pstop |
--------------------------------------------------------------------------------
| SELECT STATEMENT            |           |       3 |   54 |       1 |       |       |
|  PARTITION RANGE ITERATOR   |           |         |      |         |     4 |     5 |
|   TABLE ACCESS FULL         |EMP_RANGE  |       3 |   54 |       1 |     4 |     5 |
--------------------------------------------------------------------------------
6 rows selected.
```

In example 2a, the partition row source iterates from partition 4 to 5, because we prune the other partitions using a predicate on `hiredate`.

Example 3a

```
EXPLAIN PLAN FOR SELECT * FROM emp_range
WHERE hiredate < TO_DATE('1-JAN-1991','DD-MON-YYYY');
```

```
Plan Table
--------------------------------------------------------------------------------
| Operation                  |  Name     |  Rows | Bytes|  Cost   | Pstart| Pstop |
--------------------------------------------------------------------------------
| SELECT STATEMENT           |           |     2 |   36 |       1 |       |       |
|  TABLE ACCESS FULL         |EMP_RANGE |     2 |   36 |       1 |     1 |     1 |
--------------------------------------------------------------------------------
5 rows selected.
```

In example 3a, only partition 1 is accessed and known at compile time; thus, there is no need for a partition row source.

Plans for Hash Partitioning

Oracle displays the same information for hash partitioned objects, except that the partition row source name is `PARTITION HASH` instead of `PARTITION RANGE`. Also, with hash partitioning, pruning is only possible using equality or `IN`-list predicates.

Pruning Information with Composite Partitioned Objects

To illustrate how Oracle displays pruning information for composite partitioned objects, consider the table `emp_comp` that is range partitioned on `hiredate` and subpartitioned by hash on `deptno`.

```
CREATE TABLE emp_comp PARTITION BY RANGE(hiredate) SUBPARTITION BY HASH(deptno)
SUBPARTITIONS 3
(
    PARTITION emp_p1 VALUES LESS THAN (TO_DATE('1-JAN-1991','DD-MON-YYYY')),
    PARTITION emp_p2 VALUES LESS THAN (TO_DATE('1-JAN-1993','DD-MON-YYYY')),
    PARTITION emp_p3 VALUES LESS THAN (TO_DATE('1-JAN-1995','DD-MON-YYYY')),
    PARTITION emp_p4 VALUES LESS THAN (TO_DATE('1-JAN-1997','DD-MON-YYYY')),
    PARTITION emp_p5 VALUES LESS THAN (TO_DATE('1-JAN-1999','DD-MON-YYYY'))
)
AS SELECT * FROM emp;
```

Example 1b

```
EXPLAIN PLAN FOR SELECT * FROM emp_comp;
```

```
Plan Table
-------------------------------------------------------------------------------
| Operation                 | Name     |  Rows  | Bytes|  Cost   | Pstart | Pstop |
-------------------------------------------------------------------------------
| SELECT STATEMENT          |          |   105  |   8K|       1 |        |       |
|  PARTITION RANGE ALL      |          |        |     |         |     1  |     5 |
|   PARTITION HASH ALL      |          |        |     |         |     1  |     3 |
|    TABLE ACCESS FULL      |EMP_COMP  |   105  |   8K|       1 |     1  |     15|
-------------------------------------------------------------------------------
7 rows selected.
```

Example 1b shows the plan when Oracle accesses all subpartitions of all partitions of a composite object. Two partition row sources are used for that purpose: a range partition row source to iterate over the partitions and a hash partition row source to iterate over the subpartitions of each accessed partition.

In example 1b, because no pruning is performed, the range partition row source iterates from partition 1 to 5. Within each partition, the hash partition row source iterates over subpartitions 1 to 3 of the current partition. As a result, the table access row source accesses subpartitions 1 to 15. In other words, it accesses all subpartitions of the composite object.

Example 2b

```
EXPLAIN PLAN FOR SELECT * FROM emp_comp
WHERE hiredate = TO_DATE('15-FEB-1997', 'DD-MON-YYYY');

Plan Table
-------------------------------------------------------------------------------
| Operation                 | Name      |  Rows  | Bytes|  Cost   | Pstart| Pstop |
-------------------------------------------------------------------------------
| SELECT STATEMENT          |           |      1 |   96 |       1 |       |       |
|  PARTITION HASH ALL       |           |        |      |         |     1 |     3 |
|   TABLE ACCESS FULL       |EMP_COMP   |      1 |   96 |       1 |    13 |    15 |
-------------------------------------------------------------------------------
6 rows selected.
```

In example 2b, only the last partition, partition 5, is accessed. This partition is known at compile time, so we do not need to show it in the plan. The hash partition row source shows accessing of all subpartitions within that partition; that is, subpartitions 1 to 3, which translates into subpartitions 13 to 15 of the `emp_comp` table.

Example 3b

```
EXPLAIN PLAN FOR SELECT * FROM emp_comp WHERE deptno = 20;

Plan Table
--------------------------------------------------------------------------------
| Operation                 |  Name     |  Rows | Bytes|  Cost   | Pstart| Pstop |
--------------------------------------------------------------------------------
| SELECT STATEMENT          |           |     2 |  200 |       1 |       |       |
|  PARTITION RANGE ALL      |           |       |      |         |     1 |     5 |
|   TABLE ACCESS FULL       |EMP_COMP   |     2 |  200 |       1 |       |       |
--------------------------------------------------------------------------------
6 rows selected.
```

In example 3b, the predicate `deptno` = 20 enables pruning on the hash dimension within each partition, so Oracle only needs to access a single subpartition. The number of that subpartition is known at compile time, so the hash partition row source is not needed.

Example 4b

```
VARIABLE dno NUMBER;
EXPLAIN PLAN FOR SELECT * FROM emp_comp WHERE deptno = :dno;

Plan Table
--------------------------------------------------------------------------------
| Operation                 |  Name     |  Rows | Bytes|  Cost   | Pstart| Pstop |
--------------------------------------------------------------------------------
| SELECT STATEMENT          |           |     2 |  200 |       1 |       |       |
|  PARTITION RANGE ALL      |           |       |      |         |     1 |     5 |
|   PARTITION HASH SINGLE   |           |       |      |         |   KEY |   KEY |
|    TABLE ACCESS FULL      |EMP_COMP   |     2 |  200 |       1 |       |       |
--------------------------------------------------------------------------------
7 rows selected.
```

Example 4b is the same as example 3b, except that `deptno` = 20 has been replaced by `deptno` = :`dno`. In this case, the subpartition number is unknown at compile time, and a hash partition row source is allocated. The option is `SINGLE` for that row source, because Oracle accesses only one subpartition within each partition. The `PARTITION_START` and `PARTITION_STOP` is set to `KEY`. This means that Oracle determines the number of the subpartition at run time.

Partial Partition-wise Joins

Example 1c

In this example, `emp_range` is joined on the partitioning column and is parallelized. This enables use of partial partition-wise join, because the `dept` table is not partitioned. Oracle dynamically partitions the `dept` table before the join.

```
ALTER TABLE emp PARALLEL 2;
    STATEMENT PROCESSED.
ALTER TABLE dept PARALLEL 2;
    STATEMENT PROCESSED.
```

To show the plan for the query, enter:

```
EXPLAIN PLAN FOR SELECT /*+ ORDERED USE_HASH(D) */ ename, dname
FROM emp_range e, dept d
WHERE e.deptno = d.deptno
    AND e.hiredate > TO_DATE('29-JUN-1996','DD-MON-YYYY');
```

```
Plan Table
------------------------------------------------------------------------------------------------------------------
| Operation                 | Name      | Rows | Bytes| Cost  |  TQ   |IN-OUT| PQ Distrib | Pstart| Pstop |
------------------------------------------------------------------------------------------------------------------
| SELECT STATEMENT          |           |    1 |   51 |     3 |       |      |            |       |       |
|  HASH JOIN                |           |    1 |   51 |     3 | 2,02  | P->S |QC (RANDOM) |       |       |
|   PARTITION RANGE ITERATOR |          |      |      |       | 2,02  | PCWP |            |     4 |     5 |
|    TABLE ACCESS FULL      |EMP_RANGE  |    3 |   87 |     1 | 2,00  | PCWP |            |     4 |     5 |
|    TABLE ACCESS FULL      |DEPT       |   21 |  462 |     1 | 2,01  | P->P |PART (KEY)  |       |       |
------------------------------------------------------------------------------------------------------------------
8 rows selected.
```

The plan shows that the optimizer selects partition-wise join, because the `DIST` column contains the text `PART` (`KEY`), or partition key.

Example 2c

In example 2c, `emp_comp` is joined on its hash partitioning column, `deptno`, and is parallelized. This enables use of partial partition-wise join, because the `dept` table is not partitioned. Again, Oracle dynamically partitions the `dept` table.

```
ALTER TABLE emp_comp PARALLEL 2;
    STATEMENT PROCESSED.
EXPLAIN PLAN FOR SELECT /*+ ORDERED USE_HASH(D) */ ename, dname
    FROM emp_comp e, dept d
    WHERE e.deptno = d.deptno
    AND e.hiredate > TO_DATE('13-MAR-1995','DD-MON-YYYY');
```

```
Plan Table
-------------------------------------------------------------------------------------------------------------
| Operation                    |  Name      |  Rows | Bytes|  Cost   |  TQ  |IN-OUT| PQ Distrib | Pstart| Pstop |
-------------------------------------------------------------------------------------------------------------
| SELECT STATEMENT             |            |    1  |  51  |       3 |      |      |            |       |       |
|  HASH JOIN                   |            |     1 |   51 |       3 | 0,01 | P->S | QC (RANDOM)|       |       |
|   PARTITION RANGE ITERATOR   |            |       |      |         | 0,01 | PCWP |            |     4 |     5 |
|    PARTITION HASH ALL        |            |       |      |         | 0,01 | PCWP |            |     1 |     3 |
|     TABLE ACCESS FULL        |EMP_COMP    |     3 |   87 |       1 | 0,01 | PCWP |            |    10 |    15 |
|   TABLE ACCESS FULL          |DEPT        |    21 |  462 |       1 | 0,00 | P->P | PART (KEY) |       |       |
-------------------------------------------------------------------------------------------------------------
9 rows selected.
```

Full Partition-wise Joins

In the following example, `emp_comp` and `dept_hash` are joined on their hash partitioning columns. This enables use of full partition-wise join. The `PARTITION HASH` row source appears on top of the join row source in the plan table output.

To create the table `dept_hash`, enter:

```
CREATE TABLE dept_hash
    PARTITION BY HASH(deptno)
    PARTITIONS 3
    PARALLEL
    AS SELECT * FROM dept;
```

To show the plan for the query, enter:

```
EXPLAIN PLAN FOR SELECT /*+ ORDERED USE_HASH(D) */ ename, dname
    FROM emp_comp e, dept_hash d
    WHERE e.deptno = d.deptno
        AND e.hiredate > TO_DATE('29-JUN-1996','DD-MON-YYYY');
```

```
Plan Table
-------------------------------------------------------------------------------------------------------------
| Operation                     |  Name      |  Rows | Bytes|  Cost   |  TQ  |IN-OUT| PQ Distrib | Pstart| Pstop |
-------------------------------------------------------------------------------------------------------------
| SELECT STATEMENT              |            |     2 |   102|       2 |      |      |            |       |       |
|  PARTITION HASH ALL           |            |       |      |         | 4,00| PCWP |            |     1 |     3 |
|   HASH JOIN                   |            |     2 |  102 |       2 | 4,00| P->S | QC (RANDOM)|       |       |
|    PARTITION RANGE ITERATOR   |            |       |      |         | 4,00| PCWP |            |     4 |     5 |
|     TABLE ACCESS FULL         |EMP_COMP    |     3 |   87 |       1 | 4,00| PCWP |            |    10 |    15 |
|    TABLE ACCESS FULL          |DEPT_HASH   |    63 |    1K|       1 | 4,00| PCWP |            |     1 |     3 |
-------------------------------------------------------------------------------------------------------------
9 rows selected.
```

INLIST ITERATOR and EXPLAIN PLAN

An INLIST ITERATOR operation appears in the EXPLAIN PLAN output if an index implements an IN-list predicate. For example, for the query:

```
SELECT * FROM emp WHERE empno IN (7876, 7900, 7902);
```

The EXPLAIN PLAN output appears as follows:

```
OPERATION          OPTIONS           OBJECT_NAME
----------------   ---------------   --------------
SELECT STATEMENT
INLIST ITERATOR
TABLE ACCESS       BY ROWID          EMP
INDEX              RANGE SCAN        EMP_EMPNO
```

The INLIST ITERATOR operation iterates over the operation below it for each value in the IN-list predicate. For partitioned tables and indexes, the three possible types of IN-list columns are described in the following sections.

Index Column

If the IN-list column empno is an index column but not a partition column, then the plan is as follows (the IN-list operator appears above the table operation but below the partition operation):

```
OPERATION          OPTIONS        OBJECT_NAME   PARTITION_START   PARTITION_STOP
----------------   ------------   -----------   ---------------   --------------
SELECT STATEMENT
PARTITION          INLIST                       KEY(INLIST)       KEY(INLIST)
INLIST ITERATOR
TABLE ACCESS       BY ROWID       EMP           KEY(INLIST)       KEY(INLIST)
INDEX              RANGE SCAN     EMP_EMPNO     KEY(INLIST)       KEY(INLIST)
```

The KEY(INLIST) designation for the partition start and stop keys specifies that an IN-list predicate appears on the index start/stop keys.

Index and Partition Column

If empno is an indexed and a partition column, then the plan contains an INLIST ITERATOR operation above the partition operation:

```
OPERATION         OPTIONS        OBJECT_NAME   PARTITION_START   PARTITION_STOP
----------------  ------------   -----------   ---------------   --------------
SELECT STATEMENT
INLIST ITERATOR
PARTITION         ITERATOR                     KEY(INLIST)       KEY(INLIST)
TABLE ACCESS      BY ROWID       EMP           KEY(INLIST)       KEY(INLIST)
INDEX             RANGE SCAN     EMP_EMPNO     KEY(INLIST)       KEY(INLIST)
```

Partition Column

If `empno` is a partition column and there are no indexes, then no `INLIST ITERATOR` operation is allocated:

```
OPERATION         OPTIONS        OBJECT_NAME   PARTITION_START   PARTITION_STOP
----------------  ------------   -----------   ---------------   --------------
SELECT STATEMENT
PARTITION                                      KEY(INLIST)       KEY(INLIST)
TABLE ACCESS      BY ROWID       EMP           KEY(INLIST)       KEY(INLIST)
INDEX             RANGE SCAN     EMP_EMPNO     KEY(INLIST)       KEY(INLIST)
```

If `emp_empno` is a bitmap index, then the plan is as follows:

```
OPERATION          OPTIONS           OBJECT_NAME
----------------   ---------------   --------------
SELECT STATEMENT
INLIST ITERATOR
TABLE ACCESS       BY INDEX ROWID    EMP
BITMAP CONVERSION  TO ROWIDS
BITMAP INDEX       SINGLE VALUE      EMP_EMPNO
```

Domain Indexes and EXPLAIN PLAN

You can also use `EXPLAIN PLAN` to derive user-defined CPU and I/O costs for domain indexes. `EXPLAIN PLAN` displays these statistics in the `OTHER` column of `PLAN_TABLE`.

For example, assume table `emp` has user-defined operator `CONTAINS` with a domain index `emp_resume` on the `resume` column, and the index type of `emp_resume` supports the operator `CONTAINS`. Then the query:

```
SELECT * FROM emp WHERE CONTAINS(resume, 'Oracle') = 1
```

might display the following plan:

```
OPERATION            OPTIONS      OBJECT_NAME     OTHER
-----------------    -----------  ------------    ----------------
SELECT STATEMENT
TABLE ACCESS         BY ROWID     EMP
DOMAIN INDEX                      EMP_RESUME      CPU: 300, I/O: 4
```

EXPLAIN PLAN Restrictions

Oracle does not support EXPLAIN PLAN for statements performing implicit type conversion of date bind variables. With bind variables in general, the EXPLAIN PLAN output may not represent the real execution plan.

From the text of a SQL statement, TKPROF cannot determine the types of the bind variables. It assumes that the type is CHARACTER, and gives an error message if this is not the case. You can avoid this limitation by putting appropriate type conversions in the SQL statement.

See Also: Chapter 6, "Using SQL Trace and TKPROF".

6

Using SQL Trace and TKPROF

The SQL trace facility and TKPROF are two basic performance diagnostic tools that can help you monitor and tune applications running against the Oracle Server.

This chapter contains the following sections:

- Understanding SQL Trace and TKPROF
- Using the SQL Trace Facility and TKPROF
- Avoiding Pitfalls in TKPROF Interpretation
- TKPROF Output Example

Understanding SQL Trace and TKPROF

The SQL trace facility and TKPROF let you accurately assess the efficiency of the SQL statements your application runs. For best results, use these tools with EXPLAIN PLAN, rather than using EXPLAIN PLAN alone.

Understanding the SQL Trace Facility

The SQL trace facility provides performance information on individual SQL statements. It generates the following statistics for each statement:

- Parse, execute, and fetch counts
- CPU and elapsed times
- Physical reads and logical reads
- Number of rows processed
- Misses on the library cache
- Username under which each parse occurred
- Each commit and rollback

You can enable the SQL trace facility for a session or for an instance. When the SQL trace facility is enabled, performance statistics for all SQL statements executed in a user session or in the instance are placed into trace files.

The additional overhead of running the SQL trace facility against an application with performance problems is normally insignificant, compared with the inherent overhead caused by the application's inefficiency.

Note: Try to enable SQL trace only for statistics collection, and on specific sessions. If you must enable the facility on an entire production environment, then you can minimize performance impact with the following:

- Maintain at least 25% idle CPU capacity.
- Maintain adequate disk space for the USER_DUMP_DEST location.
- Stripe disk space over sufficient disks.

Understanding TKPROF

You can run the TKPROF program to format the contents of the trace file and place the output into a readable output file. Optionally, TKPROF can also:

- Determine the execution plans of SQL statements.
- Create a SQL script that stores the statistics in the database.

TKPROF reports each statement executed with the resources it has consumed, the number of times it was called, and the number of rows which it processed. This information lets you easily locate those statements that are using the greatest resource. With experience or with baselines available, you can assess whether the resources used are reasonable given the work done.

Using the SQL Trace Facility and TKPROF

Follow these steps to use the SQL trace facility and TKPROF:

1. Set initialization parameters for trace file management.

   See "Step 1: Setting Initialization Parameters for Trace File Management" on page 6-4.

2. Enable the SQL trace facility for the desired session, and run your application. This step produces a trace file containing statistics for the SQL statements issued by the application.

   See "Step 2: Enabling the SQL Trace Facility" on page 6-5.

3. Run TKPROF to translate the trace file created in Step 2 into a readable output file. This step can optionally create a SQL script that can be used to store the statistics in a database.

   See "Step 3: Formatting Trace Files with TKPROF" on page 6-6.

4. Interpret the output file created in Step 3.

   See "Step 4: Interpreting TKPROF Output" on page 6-11.

5. Optionally, run the SQL script produced in Step 3 to store the statistics in the database.

   See "Step 5: Storing SQL Trace Facility Statistics" on page 6-16.

In the following sections, each of these steps is discussed in depth.

Step 1: Setting Initialization Parameters for Trace File Management

When the SQL trace facility is enabled *for a session*, Oracle generates a trace file containing statistics for traced SQL statements for that session. When the SQL trace facility is enabled *for an instance*, Oracle creates a separate trace file for each process.

Before enabling the SQL trace facility, you should:

1. Check settings of the `TIMED_STATISTICS`, `MAX_DUMP_FILE_SIZE`, and `USER_DUMP_DEST` initialization parameters.

Table 6–1 SQL Trace Facility Dynamic Initialization Parameters

Parameter	Description
TIMED_STATISTICS	This enables and disables the collection of timed statistics, such as CPU and elapsed times, by the SQL trace facility, as well as the collection of various statistics in the dynamic performance tables. The default value of `false` disables timing. A value of `true` enables timing. Enabling timing causes extra timing calls for low-level operations. This is a dynamic parameter. It is also a session parameter.
MAX_DUMP_FILE_SIZE	When the SQL trace facility is enabled at the instance level, every call to the server produces a text line in a file in your operating system's file format. The maximum size of these files (in operating system blocks) is limited by this initialization parameter. The default is 500. If you find that your trace output is truncated, then increase the value of this parameter before generating another trace file. This is a dynamic parameter. It is also a session parameter.
USER_DUMP_DEST	This must fully specify the destination for the trace file according to the conventions of your operating system. The default value is the default destination for system dumps on your operating system.This value can be modified with `ALTER SYSTEM SET USER_DUMP_DEST=` *newdir*. This is a dynamic parameter. It is also a session parameter.

2. Devise a way of recognizing the resulting trace file.

   Be sure you know how to distinguish the trace files by name. Oracle writes them to the user dump destination specified by `USER_DUMP_DEST`. However, this directory may soon contain many hundreds of files, usually with generated names. It may be difficult to match trace files back to the session or process that created them. You can tag trace files by including in your programs a statement like `SELECT 'program name' FROM DUAL`. You can then trace each file back to the process that created it.

3. If your operating system retains multiple versions of files, then be sure your version limit is high enough to accommodate the number of trace files you expect the SQL trace facility to generate.
4. The generated trace files may be owned by an operating system user other than yourself. This user must make the trace files available to you before you can use TKPROF to format them.

Step 2: Enabling the SQL Trace Facility

To enable the SQL trace facility for your current *session*, enter the following:

```
ALTER SESSION SET SQL_TRACE = true;
```

Caution: Because running the SQL trace facility increases system overhead, you should enable it only when tuning your SQL statements, and disable it when you are finished.

Setting SQL_TRACE to true can have a severe performance impact. For more information, see *Oracle8i Reference*.

Alternatively, you can enable the SQL trace facility for your session by using the DBMS_SESSION.SET_SQL_TRACE procedure.

You can enable SQL trace in *another* session by using the DBMS_SYSTEM.SET_SQL_TRACE_IN_SESSION procedure.

To disable the SQL trace facility for your session, enter:

```
ALTER SESSION SET SQL_TRACE = false;
```

The SQL trace facility is automatically disabled for your session when your application disconnects from Oracle.

Note: You may need to modify your application to contain the ALTER SESSION statement. For example, to issue the ALTER SESSION statement in Oracle Forms, invoke Oracle Forms using the -s option, or invoke Oracle Forms (Design) using the statistics option. For more information on Oracle Forms, see the *Oracle Forms Reference*.

To enable the SQL trace facility for your *instance*, set the value of the SQL_TRACE initialization parameter to true. Statistics are collected for all sessions.

```
ALTER SYSTEM SET SQL_TRACE = true;
```

After the SQL trace facility has been enabled for the instance, you can disable it for the instance by entering:

```
ALTER SYSTEM SET SQL_TRACE = false;
```

Step 3: Formatting Trace Files with TKPROF

TKPROF accepts as input a trace file produced by the SQL trace facility, and it produces a formatted output file. TKPROF can also be used to generate execution plans.

After the SQL trace facility has generated a number of trace files, you can:

- Run TKPROF on each individual trace file, producing a number of formatted output files, one for each session.
- Concatenate the trace files, and then run TKPROF on the result to produce a formatted output file for the entire instance.

TKPROF does not report COMMITs and ROLLBACKs that are recorded in the trace file.

Sample TKPROF Output

Sample output from TKPROF is as follows:

```
SELECT * FROM emp, dept
WHERE emp.deptno = dept.deptno;

call   count      cpu    elapsed     disk    query current    rows
---- -------  -------  --------- -------- -------- -------  ------
Parse      1     0.16       0.29        3       13       0       0
Execute    1     0.00       0.00        0        0       0       0
Fetch      1     0.03       0.26        2        2       4      14

Misses in library cache during parse: 1
Parsing user id: (8) SCOTT

Rows     Execution Plan
-------  ---------------------------------------------------
    14  MERGE JOIN
     4   SORT JOIN
     4     TABLE ACCESS (FULL) OF 'DEPT'
    14    SORT JOIN
    14      TABLE ACCESS (FULL) OF 'EMP'
```

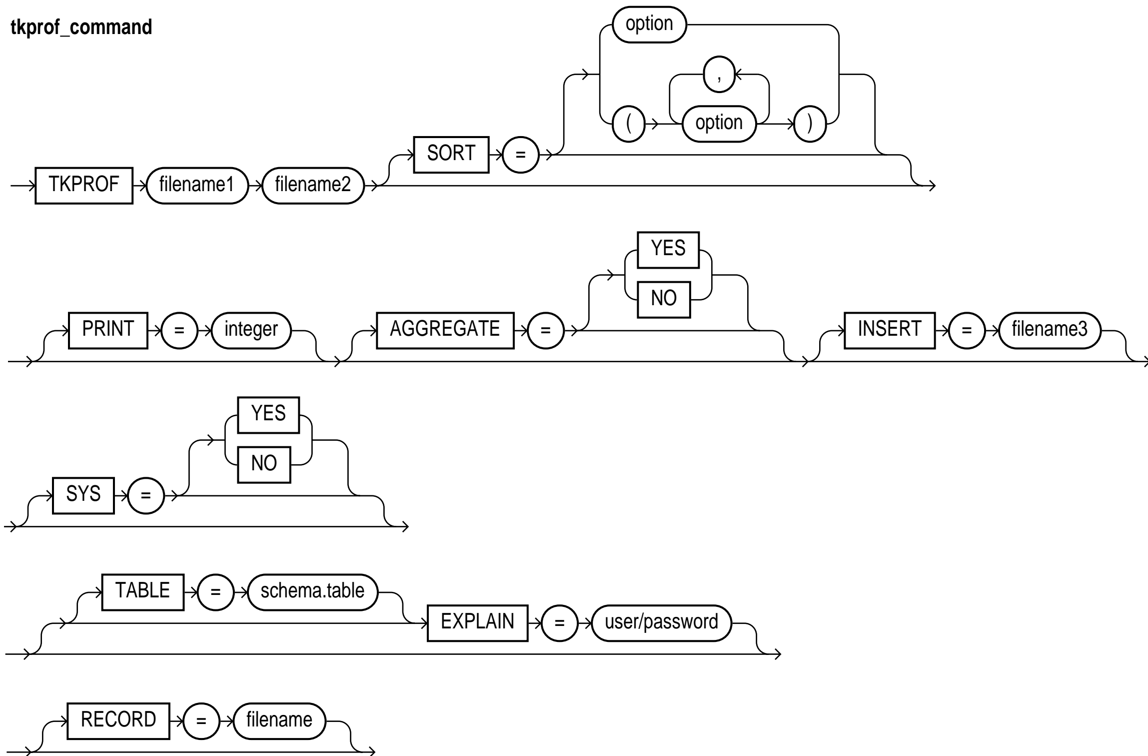
For this statement, TKPROF output includes the following information:

- The text of the SQL statement.
- The SQL trace statistics in tabular form.
- The number of library cache misses for the parsing and execution of the statement.
- The user initially parsing the statement.
- The execution plan generated by EXPLAIN PLAN.

TKPROF also provides a summary of user level statements and recursive SQL calls for the trace file.

Syntax of TKPROF

Invoke `TKPROF` using the following syntax:



If you invoke `TKPROF` without arguments, then online help is displayed.

Use the following arguments with `TKPROF`:

Table 6–2 TKPROF Arguments

Argument	Meaning
filename1	Specifies the input file, a trace file containing statistics produced by the SQL trace facility. This file can be either a trace file produced for a single session, or a file produced by concatenating individual trace files from multiple sessions.
filename2	Specifies the file to which `TKPROF` writes its formatted output.

Table 6–2 TKPROF Arguments

SORT	Sorts traced SQL statements in descending order of specified sort option before listing them into the output file. If more than one option is specified, then the output is sorted in descending order by the sum of the values specified in the sort options. If you omit this parameter, then TKPROF lists statements into the output file in order of first use. Sort options are listed below:
	PRSCNT — Number of times parsed.
	PRSCPU — CPU time spent parsing.
	PRSELA — Elapsed time spent parsing.
	PRSDSK — Number of physical reads from disk during parse.
	PRSQRY — Number of consistent mode block reads during parse.
	PRSCU — Number of current mode block reads during parse.
	PRSMIS — Number of library cache misses during parse.
	EXECNT — Number of executes.
	EXECPU — CPU time spent executing.
	EXEELA — Elapsed time spent executing.
	EXEDSK — Number of physical reads from disk during execute.
	EXEQRY — Number of consistent mode block reads during execute.
	EXECU — Number of current mode block reads during execute.
	EXEROW — Number of rows processed during execute.
	EXEMIS — Number of library cache misses during execute.
	FCHCNT — Number of fetches.
	FCHCPU — CPU time spent fetching.
	FCHELA — Elapsed time spent fetching.
	FCHDSK — Number of physical reads from disk during fetch.
	FCHQRY — Number of consistent mode block reads during fetch.
	FCHCU — Number of current mode block reads during fetch.
	FCHROW — Number of rows fetched.
PRINT	Lists only the first *integer* sorted SQL statements from the output file. If you omit this parameter, then TKPROF lists all traced SQL statements. This parameter does not affect the optional SQL script. The SQL script always generates insert data for all traced SQL statements.

Table 6–2 TKPROF Arguments

AGGREGATE	If you specify AGGREGATE = NO, then TKPROF does not aggregate multiple users of the same SQL text.
INSERT	Creates a SQL script that stores the trace file statistics in the database. TKPROF creates this script with the name filename3. This script creates a table and inserts a row of statistics for each traced SQL statement into the table.
SYS	Enables and disables the listing of SQL statements issued by the user SYS, or recursive SQL statements, into the output file. The default value of YES causes TKPROF to list these statements. The value of NO causes TKPROF to omit them. This parameter does not affect the optional SQL script. The SQL script always inserts statistics for all traced SQL statements, including recursive SQL statements.
TABLE	Specifies the *schema* and name of the *table* into which TKPROF temporarily places execution plans before writing them to the output file. If the specified table already exists, then TKPROF deletes all rows in the table, uses it for the EXPLAIN PLAN statement (which writes more rows into the table), and then deletes those rows. If this table does not exist, then TKPROF creates it, uses it, and then drops it. The specified *user* must be able to issue INSERT, SELECT, and DELETE statements against the table. If the table does not already exist, then the user must also be able to issue CREATE TABLE and DROP TABLE statements. For the privileges to issue these statements, see the *Oracle8i SQL Reference.* This option allows multiple individuals to run TKPROF concurrently with the same *user* in the EXPLAIN value. These individuals can specify different TABLE values and avoid destructively interfering with each other's processing on the temporary plan table. If you use the EXPLAIN parameter without the TABLE parameter, then TKPROF uses the table PROF$PLAN_TABLE in the schema of the *user* specified by the EXPLAIN parameter. If you use the TABLE parameter without the EXPLAIN parameter, then TKPROF ignores the TABLE parameter.
EXPLAIN	Determines the execution plan for each SQL statement in the trace file and writes these execution plans to the output file. TKPROF determines execution plans by issuing the EXPLAIN PLAN statement after connecting to Oracle with the *user* and *password* specified in this parameter. The specified *user* must have CREATE SESSION system privileges. TKPROF takes longer to process a large trace file if the EXPLAIN option is used.
RECORD	Creates a SQL script with the specified filename with all of the nonrecursive SQL in the trace file. This can be used to replay the user events from the trace file.

TKPROF Statement Examples

This section provides two brief examples of TKPROF usage. For an complete example of TKPROF output, see "TKPROF Output Example" on page 6-22.

Example 1 If you are processing a large trace file using a combination of SORT parameters and the PRINT parameter, then you can produce a TKPROF output file

containing only the highest resource-intensive statements. For example, the following statement prints the ten statements in the trace file that have generated the most physical I/O:

```
TKPROF ora53269.trc ora53269.prf SORT = (PRSDSK, EXEDSK, FCHDSK) PRINT = 10
```

Example 2 This example runs TKPROF, accepts a trace file named dlsun12_jane_fg_sqlplus_007.trc, and writes a formatted output file named outputa.prf:

```
TKPROF dlsun12_jane_fg_sqlplus_007.trc OUTPUTA.PRF
EXPLAIN=scott/tiger TABLE=scott.temp_plan_table_a INSERT=STOREA.SQL SYS=NO
SORT=(EXECPU,FCHCPU)
```

This example is likely to be longer than a single line on your screen and you may need to use continuation characters, depending on your operating system.

Note the other parameters in this example:

- The EXPLAIN value causes TKPROF to connect as the user scott and use the EXPLAIN PLAN statement to generate the execution plan for each traced SQL statement. You can use this to get access paths and row source counts.
- The TABLE value causes TKPROF to use the table temp_plan_table_a in the schema scott as a temporary plan table.
- The INSERT value causes TKPROF to generate a SQL script named STOREA.SQL that stores statistics for all traced SQL statements in the database.
- The SYS parameter with the value of NO causes TKPROF to omit recursive SQL statements from the output file. In this way you can ignore internal Oracle statements such as temporary table operations.
- The SORT value causes TKPROF to sort the SQL statements in order of the sum of the CPU time spent executing and the CPU time spent fetching rows before writing them to the output file. For greatest efficiency, always use SORT parameters.

Step 4: Interpreting TKPROF Output

This section provides pointers for interpreting TKPROF output.

- Tabular Statistics
- Library Cache Misses
- Statement Truncation

- User Issuing the SQL Statement
- Execution Plan
- Deciding Which Statements to Tune

While TKPROF provides a very useful analysis, the most accurate measure of efficiency is the actual performance of the application in question. At the end of the TKPROF output is a summary of the work done in the database engine by the process during the period that the trace was running.

Tabular Statistics

TKPROF lists the statistics for a SQL statement returned by the SQL trace facility in rows and columns. Each row corresponds to one of three steps of SQL statement processing. Statistics are identified by the value of the CALL column:

PARSE	This translates the SQL statement into an execution plan, including checks for proper security authorization and checks for the existence of tables, columns, and other referenced objects.
EXECUTE	This is the actual execution of the statement by Oracle. For INSERT, UPDATE, and DELETE statements, this modifies the data. For SELECT statements, this identifies the selected rows.
FETCH	This retrieves rows returned by a query. Fetches are only performed for SELECT statements.

The other columns of the SQL trace facility output are combined statistics for all parses, all executes, and all fetches of a statement. The sum of query and current is the total number of buffers accessed.

COUNT	Number of times a statement was parsed, executed, or fetched.
CPU	Total CPU time in seconds for all parse, execute, or fetch calls for the statement. This value is zero (0) if TIMED_STATISTICS is not turned on.
ELAPSED	Total elapsed time in seconds for all parse, execute, or fetch calls for the statement. This value is zero (0) if TIMED_STATISTICS is not turned on.
DISK	Total number of data blocks physically read from the datafiles on disk for all parse, execute, or fetch calls.

`QUERY`	Total number of buffers retrieved in consistent mode for all parse, execute, or fetch calls. Buffers are usually retrieved in consistent mode for queries.
`CURRENT`	Total number of buffers retrieved in current mode. Buffers are retrieved in current mode for statements such as `INSERT`, `UPDATE`, and `DELETE`.

Statistics about the processed rows appear in the `ROWS` column.

`ROWS`	Total number of rows processed by the SQL statement. This total does not include rows processed by subqueries of the SQL statement.

For `SELECT` statements, the number of rows returned appears for the fetch step. For `UPDATE`, `DELETE`, and `INSERT` statements, the number of rows processed appears for the execute step.

Note: The row source counts are displayed when a cursor is closed. In SQL*Plus, there is only one user cursor, so each statement executed causes the previous cursor to be closed; for this reason, the row source counts are displayed. PL/SQL has its own cursor handling and does not close child cursors when the parent cursor is closed. Exiting (or reconnecting) causes the counts to be displayed.

Resolution of Statistics

Timing statistics have a resolution of one hundredth of a second; therefore, any operation on a cursor that takes a hundredth of a second or less may not be timed accurately. Keep this in mind when interpreting statistics. In particular, be careful when interpreting the results from simple queries that execute very quickly.

Recursive Calls

Sometimes, in order to execute a SQL statement issued by a user, Oracle must issue additional statements. Such statements are called *recursive calls* or *recursive SQL statements*. For example, if you insert a row into a table that does not have enough space to hold that row, then Oracle makes recursive calls to allocate the space dynamically. Recursive calls are also generated when data dictionary information is not available in the data dictionary cache and must be retrieved from disk.

If recursive calls occur while the SQL trace facility is enabled, then TKPROF produces statistics for the recursive SQL statements and marks them clearly as recursive SQL statements in the output file. You can suppress the listing of Oracle internal recursive calls (e.g., space management) in the output file by setting the SYS command-line parameter to NO. The statistics for a recursive SQL statement are included in the listing for that statement, not in the listing for the SQL statement that caused the recursive call. So, when you are calculating the total resources required to process a SQL statement, you should consider the statistics for that statement as well as those for recursive calls caused by that statement.

Note: Recursive SQL statistics are not included for SQL-level operations. However, recursive SQL statistics *are* included for operations done below the SQL level, such as triggers. For more information, see "The Trigger Trap" on page 6-22.

Library Cache Misses

TKPROF also lists the number of library cache misses resulting from parse and execute steps for each SQL statement. These statistics appear on separate lines following the tabular statistics. If the statement resulted in no library cache misses, then TKPROF does not list the statistic. In "Sample TKPROF Output" on page 6-6, the statement resulted in one library cache miss for the parse step, and no misses for the execute step.

Statement Truncation

The following SQL statements are truncated to 25 characters in the SQL trace file:

```
SET ROLE
GRANT
ALTER USER
ALTER ROLE
CREATE USER
CREATE ROLE
```

User Issuing the SQL Statement

TKPROF also lists the user ID of the user issuing each SQL statement. If the SQL trace input file contained statistics from multiple users and the statement was issued by more than one user, then TKPROF lists the ID of the last user to parse the statement. The user ID of all database users appears in the data dictionary in the column ALL_USERS.USER_ID.

Execution Plan

If you specify the EXPLAIN parameter on the TKPROF statement line, then TKPROF uses the EXPLAIN PLAN statement to generate the execution plan of each SQL statement traced. TKPROF also displays the number of rows processed by each step of the execution plan.

Note: Trace files generated immediately after instance startup contain data that reflects the activity of the startup process. In particular, they reflect a disproportionate amount of I/O activity as caches in the system global area (SGA) are filled. For the purposes of tuning, ignore such trace files.

See Also: Chapter 5, "Using EXPLAIN PLAN" has more information on interpreting execution plans.

Deciding Which Statements to Tune

You need to find which SQL statements use the most CPU or disk resource.

If the TIMED_STATISTICS parameter is on, then you can find high CPU activity in the CPU column. If TIMED_STATISTICS is not on, then check the QUERY and CURRENT columns.

See Also: For examples of finding resource intensive statements, see "TKPROF Statement Examples" on page 6-10.

With the exception of locking problems and inefficient PL/SQL loops, neither the CPU time nor the elapsed time are necessary to find problem statements. The key is the number of block visits, both query (that is, subject to read consistency) and current (that is, *not* subject to read consistency). Segment headers and blocks that are going to be updated are always acquired in current mode, but all query and subquery processing requests the data in query mode. These are precisely the same measures as the instance statistics CONSISTENT GETS and DB BLOCK GETS.

You can find high disk activity in the disk column.

The following listing shows TKPROF output for one SQL statement as it appears in the output file:

```
SELECT *
FROM emp, dept
WHERE emp.deptno = dept.deptno;

call   count      cpu    elapsed     disk    query current    rows
---- -------  -------  --------- -------- -------- -------  ------
Parse     11     0.08       0.18        0        0       0       0
Execute   11     0.23       0.66        0        3       6       0
Fetch     35     6.70       6.83      100    12326       2     824
------------------------------------------------------------------
total     57     7.01       7.67      100    12329       8     826

Misses in library cache during parse: 0
```

If it is acceptable to have 7.01 CPU seconds and to retrieve 824 rows, then you need not look any further at this trace output. In fact, a major use of TKPROF reports in a tuning exercise is to eliminate processes from the detailed tuning phase.

You can also see that 10 unnecessary parse call were made (because there were 11 parse calls for this one statement) and that array fetch operations were performed. You know this because more rows were fetched than there were fetches performed.

Step 5: Storing SQL Trace Facility Statistics

You may want to keep a history of the statistics generated by the SQL trace facility for your application, and compare them over time. TKPROF can generate a SQL script that creates a table and inserts rows of statistics into it. This script contains:

- A CREATE TABLE statement that creates an output table named TKPROF_TABLE.
- INSERT statements that add rows of statistics, one for each traced SQL statement, to the TKPROF_TABLE.

After running TKPROF, you can run this script to store the statistics in the database.

Generating the TKPROF Output SQL Script

When you run TKPROF, use the INSERT parameter to specify the name of the generated SQL script. If you omit this parameter, then TKPROF does not generate a script.

Editing the TKPROF Output SQL Script

After TKPROF has created the SQL script, you may want to edit the script before running it. If you have already created an output table for previously collected statistics and you want to add new statistics to this table, then remove the CREATE TABLE statement from the script. The script then inserts the new rows into the existing table.

If you have created multiple output tables, perhaps to store statistics from different databases in different tables, then edit the CREATE TABLE and INSERT statements to change the name of the output table.

Querying the Output Table

The following CREATE TABLE statement creates the TKPROF_TABLE:

```
CREATE TABLE TKPROF_TABLE (
    DATE_OF_INSERT    DATE,
    CURSOR_NUM        NUMBER,
    DEPTH             NUMBER,
    USER_ID           NUMBER,
    PARSE_CNT         NUMBER,
    PARSE_CPU         NUMBER,
    PARSE_ELAP        NUMBER,
    PARSE_DISK        NUMBER,
    PARSE_QUERY       NUMBER,
    PARSE_CURRENT     NUMBER,
    PARSE_MISS        NUMBER,
    EXE_COUNT         NUMBER,
    EXE_CPU           NUMBER,
    EXE_ELAP          NUMBER,
    EXE_DISK          NUMBER,
    EXE_QUERY         NUMBER,
    EXE_CURRENT       NUMBER,
    EXE_MISS          NUMBER,
    EXE_ROWS          NUMBER,
    FETCH_COUNT       NUMBER,
    FETCH_CPU         NUMBER,
    FETCH_ELAP        NUMBER,
    FETCH_DISK        NUMBER,
    FETCH_QUERY       NUMBER,
    FETCH_CURRENT     NUMBER,
    FETCH_ROWS        NUMBER,
    CLOCK_TICKS       NUMBER,
    SQL_STATEMENT     LONG);
```

Most output table columns correspond directly to the statistics that appear in the formatted output file. For example, the PARSE_CNT column value corresponds to the count statistic for the parse step in the output file.

These columns help you identify a row of statistics:

SQL_STATEMENT — This is the SQL statement for which the SQL trace facility collected the row of statistics. Because this column has datatype LONG, you cannot use it in expressions or WHERE clause conditions.

DATE_OF_INSERT — This is the date and time when the row was inserted into the table. This value is not exactly the same as the time the statistics were collected by the SQL trace facility.

DEPTH — This indicates the level of recursion at which the SQL statement was issued. For example, a value of 0 indicates that a user issued the statement. A value of 1 indicates that Oracle generated the statement as a recursive call to process a statement with a value of 0 (a statement issued by a user). A value of *n* indicates that Oracle generated the statement as a recursive call to process a statement with a value of *n*-1.

USER_ID — This identifies the user issuing the statement. This value also appears in the formatted output file.

CURSOR_NUM — Oracle uses this column value to keep track of the cursor to which each SQL statement was assigned.

The output table does not store the statement's execution plan. The following query returns the statistics from the output table. These statistics correspond to the formatted output shown in the section "Sample TKPROF Output" on page 6-6.

```
SELECT * FROM TKPROF_TABLE;
```

Oracle responds with something similar to:

```
DATE_OF_INSERT CURSOR_NUM DEPTH USER_ID PARSE_CNT PARSE_CPU PARSE_ELAP
-------------- ---------- ----- ------- --------- --------- ----------
21-DEC-1998             1     0      8          1        16         22

PARSE_DISK PARSE_QUERY PARSE_CURRENT PARSE_MISS EXE_COUNT EXE_CPU
---------- ----------- ------------- ---------- --------- -------
    3          11            0             1           1        0
```

```
EXE_ELAP EXE_DISK EXE_QUERY EXE_CURRENT EXE_MISS EXE_ROWS FETCH_COUNT
-------- -------- --------- ----------- -------- -------- -----------
    0        0        0           0         0        0          1

FETCH_CPU FETCH_ELAP FETCH_DISK FETCH_QUERY FETCH_CURRENT FETCH_ROWS
--------- ---------- ---------- ----------- ------------- ----------
     2        20          2           2            4            10

SQL_STATEMENT
---------------------------------------------------------------------
SELECT * FROM EMP, DEPT WHERE EMP.DEPTNO = DEPT.DEPTNO
```

Avoiding Pitfalls in TKPROF Interpretation

This section describes some fine points of TKPROF interpretation:

- The Argument Trap
- The Read Consistency Trap
- The Schema Trap
- The Time Trap
- The Trigger Trap

The Argument Trap

If you are not aware of the values being bound at run time, then it is possible to fall into the "argument trap". EXPLAIN PLAN cannot determine the type of a bind variable from the text of SQL statements, and it always assumes that the type is varchar. If the bind variable is actually a number or a date, then TKPROF can cause implicit data conversions, which can cause inefficient plans to be executed. To avoid this, you should experiment with different data types in your query.

> **See Also:** "EXPLAIN PLAN Restrictions" on page 5-23 has information about TKPROF and bind variables.

The Read Consistency Trap

The next example illustrates the read consistency trap. Without knowing that an uncommitted transaction had made a series of updates to the NAME column it is very difficult to see why so many block visits would be incurred.

Cases like this are not normally repeatable: if the process were run again, it is unlikely that another transaction would interact with it in the same way.

```
SELECT name_id
FROM cq_names
WHERE name = 'FLOOR';

call     count     cpu     elapsed     disk     query current     rows
----     -----     ---     -------     ----     ----- -------     ----
Parse        1    0.10        0.18        0         0       0        0
Execute      1    0.00        0.00        0         0       0        0
Fetch        1    0.11        0.21        2       101       0        1

Misses in library cache during parse: 1
Parsing user id: 01 (USER1)

Rows     Execution Plan
----     --------- ----
   0     SELECT STATEMENT
   1       TABLE ACCESS (BY ROWID) OF 'CQ_NAMES'
   2         INDEX (RANGE SCAN) OF 'CQ_NAMES_NAME' (NON_UNIQUE)
```

The Schema Trap

This example shows an extreme (and thus easily detected) example of the schema trap. At first, it is difficult to see why such an apparently straightforward indexed query needs to look at so many database blocks, or why it should access any blocks at all in current mode.

```
SELECT name_id
FROM cq_names
WHERE name = 'FLOOR';

call        count        cpu      elapsed     disk  query current rows
--------  -------    --------    ---------  ------- ------ ------- ----
Parse           1       0.06         0.10        0       0       0    0
Execute         1       0.02         0.02        0       0       0    0
Fetch           1       0.23         0.30       31      31       3    1

Misses in library cache during parse: 0
Parsing user id: 02  (USER2)
```

```
Rows     Execution Plan
-------  ---------------------------------------------------
      0  SELECT STATEMENT
   2340    TABLE ACCESS (BY ROWID) OF 'CQ_NAMES'
      0      INDEX (RANGE SCAN) OF 'CQ_NAMES_NAME' (NON-UNIQUE)
```

Two statistics suggest that the query may have been executed with a full table scan. These statistics are the current mode block visits, plus the number of rows originating from the Table Access row source in the execution plan. The explanation is that the required index was built after the trace file had been produced, but before `TKPROF` had been run.

Generating a new trace file gives the data below:

```
SELECT name_id
FROM cq_names
WHERE name = 'FLOOR';

call    count    cpu   elapsed  disk  query current     rows
-----  ------ ------  -------- ----- ------ -------    -----
Parse       1   0.01      0.02     0      0       0        0
Execute     1   0.00      0.00     0      0       0        0
Fetch       1   0.00      0.00     0      2       0        1

Misses in library cache during parse: 0
Parsing user id: 02  (USER2)

Rows     Execution Plan
-------  ---------------------------------------------------
      0  SELECT STATEMENT
      1    TABLE ACCESS (BY ROWID) OF 'CQ_NAMES'
      2      INDEX (RANGE SCAN) OF 'CQ_NAMES_NAME' (NON-UNIQUE)
```

One of the marked features of this correct version is that the parse call took 10 milliseconds of CPU time and 20 milliseconds of elapsed time, but the query apparently took no time at all to execute and perform the fetch. These anomalies arise because the clock tick of 10 milliseconds is too long relative to the time taken to execute and fetch the data. In such cases, it is important to get lots of executions of the statements, so that you have statistically valid numbers.

The Time Trap

Sometimes, as in the following example, you may wonder why a particular query has taken so long.

```
UPDATE cq_names SET ATTRIBUTES = lower(ATTRIBUTES)
WHERE ATTRIBUTES = :att

call       count       cpu    elapsed     disk    query current        rows
-------- -------  --------  --------- -------- -------- -------  ----------
Parse          1      0.06       0.24        0        0       0           0
Execute        1      0.62      19.62       22      526      12           7
Fetch          0      0.00       0.00        0        0       0           0

Misses in library cache during parse: 1
Parsing user id: 02  (USER2)

Rows     Execution Plan
-------  ---------------------------------------------------
      0  UPDATE STATEMENT
   2519  TABLE ACCESS (FULL) OF 'CQ_NAMES'
```

Again, the answer is interference from another transaction. In this case, another transaction held a shared lock on the table `cq_names` for several seconds before and after the update was issued. It takes a fair amount of experience to diagnose that interference effects are occurring. On the one hand, comparative data is essential when the interference is contributing only a short delay (or a small increase in block visits in the previous example). On the other hand, if the interference is contributing only a modest overhead, and the statement is essentially efficient, then its statistics may never have to be subjected to analysis.

The Trigger Trap

The resources reported for a statement include those for all of the SQL issued while the statement was being processed. Therefore, they include any resources used within a trigger, along with the resources used by any other recursive SQL (such as that used in space allocation). With the SQL trace facility enabled, `TKPROF` reports these resources twice. Avoid trying to tune the DML statement if the resource is actually being consumed at a lower level of recursion.

You may need to inspect the raw trace file to see exactly where the resource is being expended. The entries for recursive SQL follow the `PARSING IN CURSOR` entry for the user's statement. Within the trace file, the order is less easily defined.

TKPROF Output Example

This section provides an extensive example of `TKPROF` output. Portions have been edited out for the sake of brevity.

Header

```
Copyright (c) Oracle Corporation 1979, 1999. All rights reserved.
Trace file: v80_ora_2758.trc
Sort options: default
********************************************************************************
count    = number of times OCI procedure was executed
cpu      = cpu time in seconds executing
elapsed  = elapsed time in seconds executing
disk     = number of physical reads of buffers from disk
query    = number of buffers gotten for consistent read
current  = number of buffers gotten in current mode (usually for update)
rows     = number of rows processed by the fetch or execute call
********************************************************************************
The following statement encountered a error during parse:
select deptno, avg(sal) from emp e group by deptno
       having exists (select deptno from dept
      where dept.deptno = e.deptno
      and dept.budget > avg(e.sal)) order by 1
Error encountered: ORA-00904
********************************************************************************
```

Body

```
ALTER SESSION SET SQL_TRACE = true
call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        0      0.00       0.00          0          0          0           0
Execute      1      0.00       0.10          0          0          0           0
Fetch        0      0.00       0.00          0          0          0           0
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total        1      0.00       0.10          0          0          0           0
Misses in library cache during parse: 0
Misses in library cache during execute: 1
Optimizer goal: CHOOSE
Parsing user id: 02   (USER02)
********************************************************************************
SELECT emp.ename, dept.dname
FROM emp, dept
  WHERE emp.deptno = dept.deptno

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.11       0.13          2          0          1           0
Execute      1      0.00       0.00          0          0          0           0
```

```
Fetch        1      0.00       0.00          2          2          4          14
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total        3      0.11       0.13          4          2          5          14
Misses in library cache during parse: 1
Optimizer goal: CHOOSE
Parsing user id: 02   (USER02)
Rows     Execution Plan
-------  ---------------------------------------------------
      0  SELECT STATEMENT   GOAL: CHOOSE
     14   MERGE JOIN
      4    SORT (JOIN)
      4     TABLE ACCESS (FULL) OF 'DEPT'
     14    SORT (JOIN)
     14     TABLE ACCESS (FULL) OF 'EMP'

********************************************************************************
SELECT a.ename name, b.ename manager
FROM emp a, emp b
  WHERE a.mgr = b.empno(+)

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.01       0.01          0          0          0           0
Execute      1      0.00       0.00          0          0          0           0
Fetch        1      0.01       0.01          1         50          2          14
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total        3      0.02       0.02          1         50          2          14
Misses in library cache during parse: 1
Optimizer goal: CHOOSE
Parsing user id: 01  (USER01)
Rows     Execution Plan
-------  ---------------------------------------------------
      0  SELECT STATEMENT   GOAL: CHOOSE
     13   NESTED LOOPS (OUTER)
     14    TABLE ACCESS (FULL) OF 'EMP'
     13    TABLE ACCESS (BY ROWID) OF 'EMP'
    26     INDEX (RANGE SCAN) OF 'EMP_IND' (NON-UNIQUE)
********************************************************************************
SELECT ename, job, sal
FROM emp
WHERE sal =
       (SELECT max(sal)
        FROM emp)

call     count       cpu    elapsed       disk      query    current        rows
```

```
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.00       0.00          0          0          0           0
Execute      1      0.00       0.00          0          0          0           0
Fetch        1      0.00       0.00          0         12          4           1
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total        3      0.00       0.00          0         12          4           1
Misses in library cache during parse: 1
Optimizer goal: CHOOSE
Parsing user id: 01  (USER01)
Rows     Execution Plan
-------  ---------------------------------------------------
      0  SELECT STATEMENT   GOAL: CHOOSE
     14   FILTER
     14    TABLE ACCESS (FULL) OF 'EMP'
     14    SORT (AGGREGATE)
     14     TABLE ACCESS (FULL) OF 'EMP'
********************************************************************************
SELECT deptno
FROM emp
WHERE job = 'clerk'
GROUP BY deptno
HAVING COUNT(*) >= 2

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.00       0.00          0          0          0           0
Execute      1      0.00       0.00          0          0          0           0
Fetch        1      0.00       0.00          0          1          1           0
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total        3      0.00       0.00          0          1          1           0
Misses in library cache during parse: 13
Optimizer goal: CHOOSE
Parsing user id: 01  (USER01)
Rows     Execution Plan
-------  ---------------------------------------------------
      0  SELECT STATEMENT   GOAL: CHOOSE
      0   FILTER
      0    SORT (GROUP BY)
     14     TABLE ACCESS (FULL) OF 'EMP'
********************************************************************************
SELECT dept.deptno, dname, job, ename
FROM dept,emp
WHERE dept.deptno = emp.deptno(+)
ORDER BY dept.deptno
```

```
call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.00       0.00          0          0          0           0
Execute      1      0.00       0.00          0          0          0           0
Fetch        1      0.00       0.00          0          3          3          10
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total        3      0.00       0.00          0          3          3          10
Misses in library cache during parse: 1
Optimizer goal: CHOOSE
Parsing user id: 01 (USER01)
Rows     Execution Plan
-------  ---------------------------------------------------
      0  SELECT STATEMENT   GOAL: CHOOSE
     14   MERGE JOIN (OUTER)
      4    SORT (JOIN)
      4     TABLE ACCESS (FULL) OF 'DEPT'
     14    SORT (JOIN)
     14     TABLE ACCESS (FULL) OF 'EMP'
********************************************************************************
SELECT grade, job, ename, sal
FROM emp, salgrade
WHERE sal BETWEEN losal AND hisal
ORDER BY grade, job

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.04       0.06          2         16          1           0
Execute      1      0.00       0.00          0          0          0           0
Fetch        1      0.01       0.01          1         10         12          10
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total        3      0.05       0.07          3         26         13          10
Misses in library cache during parse: 1
Optimizer goal: CHOOSE
Parsing user id: 02   (USER02)
Rows     Execution Plan
-------  ---------------------------------------------------
      0  SELECT STATEMENT   GOAL: CHOOSE
     14   SORT (ORDER BY)
     14    NESTED LOOPS
      5     TABLE ACCESS (FULL) OF 'SALGRADE'
     70     TABLE ACCESS (FULL) OF 'EMP'
********************************************************************************
SELECT LPAD(' ',level*2)||ename org_chart, level, empno, mgr, job, deptno
FROM emp
CONNECT BY prior empno = mgr
```

```
START WITH ename = 'clark'
  OR ename = 'blake'
ORDER BY deptno

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.01       0.01          0          0          0           0
Execute      1      0.00       0.00          0          0          0           0
Fetch        1      0.01       0.01          0          1          2           0
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total        3      0.02       0.02          0          1          2           0
Misses in library cache during parse: 1
Optimizer goal: CHOOSE
Parsing user id: 02   (USER02)
Rows     Execution Plan
-------  ---------------------------------------------------
      0  SELECT STATEMENT   GOAL: CHOOSE
      0   SORT (ORDER BY)
      0    CONNECT BY
     14     TABLE ACCESS (FULL) OF 'EMP'
      0     TABLE ACCESS (BY ROWID) OF 'EMP'
      0     TABLE ACCESS (FULL) OF 'EMP'
********************************************************************************
CREATE TABLE TKOPTKP (a number, b number)

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.00       0.00          0          0          0           0
Execute      1      0.01       0.01          1          0          1           0
Fetch        0      0.00       0.00          0          0          0           0
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total        2      0.01       0.01          1          0          1           0
Misses in library cache during parse: 1
Optimizer goal: CHOOSE
Parsing user id: 02   (USER02)
Rows     Execution Plan
-------  ---------------------------------------------------
      0  CREATE TABLE STATEMENT   GOAL: CHOOSE

********************************************************************************
INSERT INTO TKOPTKP
VALUES (1,1)

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
```

```
Parse        1      0.07       0.09          0          0          0           0
Execute      1      0.01       0.20          2          2          3           1
Fetch        0      0.00       0.00          0          0          0           0
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total        2      0.08       0.29          2          2          3           1
Misses in library cache during parse: 1
Optimizer goal: CHOOSE
Parsing user id: 02   (USER02)
Rows     Execution Plan
-------  ---------------------------------------------------
      0  INSERT STATEMENT   GOAL: CHOOSE

********************************************************************************
INSERT INTO TKOPTKP SELECT * FROM TKOPTKP

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.00       0.00          0          0          0           0
Execute      1      0.02       0.02          0          2          3          11
Fetch        0      0.00       0.00          0          0          0           0
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total        2      0.02       0.02          0          2          3          11
Misses in library cache during parse: 1
Optimizer goal: CHOOSE
Parsing user id: 02   (USER02)
Rows     Execution Plan
-------  ---------------------------------------------------
      0  INSERT STATEMENT   GOAL: CHOOSE
     12   TABLE ACCESS (FULL) OF 'TKOPTKP'
********************************************************************************
SELECT *
FROM TKOPTKP
WHERE a > 2

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.01       0.01          0          0          0           0
Execute      1      0.00       0.00          0          0          0           0
Fetch        1      0.00       0.00          0          1          2          10
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total        3      0.01       0.01          0          1          2          10
Misses in library cache during parse: 1
Optimizer goal: CHOOSE
Parsing user id: 02   (USER02)
Rows     Execution Plan
```

```
-------  --------------------------------------------------
      0  SELECT STATEMENT   GOAL: CHOOSE
     24   TABLE ACCESS (FULL) OF 'TKOPTKP'
********************************************************************************
```

Summary

```
OVERALL TOTALS FOR ALL NON-RECURSIVE STATEMENTS
call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse       18      0.40       0.53         30        182          3           0
Execute     19      0.05       0.41          3          7         10          16
Fetch       12      0.05       0.06          4        105         66          78
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total       49      0.50       1.00         37        294         79          94
Misses in library cache during parse: 18
Misses in library cache during execute: 1

OVERALL TOTALS FOR ALL RECURSIVE STATEMENTS
call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse       69      0.49       0.60          9         12          8           0
Execute    103      0.13       0.54          0          0          0           0
Fetch      213      0.12       0.27         40        435          0         162
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total      385      0.74       1.41         49        447          8         162
Misses in library cache during parse: 13
   19  user  SQL statements in session.
   69  internal SQL statements in session.
   88  SQL statements in session.
   17  statements EXPLAINed in this session.
********************************************************************************
Trace file: v80_ora_2758.trc
Trace file compatibility: 7.03.02
Sort options: default
       1  session in tracefile.
      19  user  SQL statements in trace file.
      69  internal SQL statements in trace file.
      88  SQL statements in trace file.
      41  unique SQL statements in trace file.
      17  SQL statements EXPLAINed using schema:
           SCOTT.prof$plan_table
             Default table was used.
             Table was created.
             Table was dropped.
```

```
1017  lines in trace file.
```

7 Using Optimizer Hints

This chapter offers recommendations on how to use cost-based optimizer hints to enhance Oracle performance.

This chapter contains the following sections:

- Understanding Hints
- Using Hints

Understanding Hints

As an application designer, you may know information about your data that the optimizer does not know. For example, you may know that a certain index is more selective for certain queries. Based on this information, you may be able to choose a more efficient execution plan than the optimizer. In such a case, use hints to force the optimizer to use the optimal execution plan.

Hints allow you to make decisions usually made by the optimizer. You can use hints to specify the following:

- The optimization approach for a SQL statement.
- The goal of the cost-based optimizer for a SQL statement.
- The access path for a table accessed by the statement.
- The join order for a join statement.
- A join operation in a join statement.

Note: The use of hints involves extra code that must also be managed, checked, and controlled.

Hints provide a mechanism to direct the optimizer to choose a certain query execution plan based on the following criteria:

- Join order
- Join method
- Access method
- Parallelization

Hints (except for the `RULE` hint) invoke the cost-based optimizer (CBO). If you have not gathered statistics, then defaults are used.

See Also: For more information on default values, see Chapter 8, "Gathering Statistics".

Specifying Hints

Hints apply only to the optimization of the statement block in which they appear. A statement block is any one of the following statements or parts of statements:

- A simple `SELECT`, `UPDATE`, or `DELETE` statement.

- A parent statement or subquery of a complex statement.
- A part of a compound query.

For example, a compound query consisting of two component queries combined by the `UNION` operator has two statement blocks, one for each component query. For this reason, hints in the first component query apply only to its optimization, not to the optimization of the second component query.

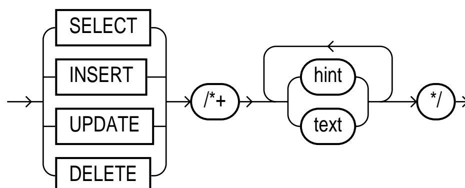
You can send hints for a SQL statement to the optimizer by enclosing them in a comment within the statement.

> **See Also:** For more information on comments, see *Oracle8i SQL Reference.*

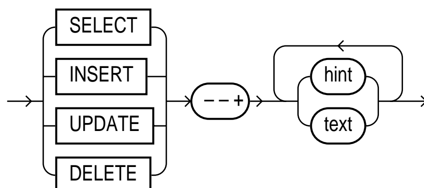
A statement block can have only one comment containing hints. This comment can only follow the `SELECT`, `UPDATE`, or `DELETE` keyword.

> **Exception:** The `APPEND` hint follows the `INSERT` keyword.

The syntax diagrams show the syntax for hints contained in both styles of comments that Oracle supports within a statement block.



or:



where:

`DELETE` `SELECT` `UPDATE`	Is a keyword that begins a statement block. Comments containing hints can appear only after these keywords.
`+`	Causes Oracle to interpret the comment as a list of hints. The plus sign must immediately follow the comment delimiter (no space is permitted).
`hint`	Is one of the hints discussed in this section. If the comment contains multiple hints, then each pair of hints must be separated by at least one space.
`text`	Is other commenting text that can be interspersed with the hints.

If you specify hints incorrectly, then Oracle ignores them, but does not return an error:

- Oracle ignores hints if the comment containing them does not follow a `DELETE`, `SELECT`, or `UPDATE` keyword.
- Oracle ignores hints containing syntax errors, but considers other correctly specified hints within the same comment.
- Oracle ignores combinations of conflicting hints, but considers other hints within the same comment.
- Oracle ignores hints in all SQL statements in those environments that use PL/SQL version 1, such as Forms version 3 triggers, Oracle Forms 4.5, and Oracle Reports 2.5. Note: These hints can be passed to the server, but the server ignores them.

Other conditions specific to index type appear later in this chapter.

The optimizer recognizes hints only when using the cost-based approach. If you include a hint (except the `RULE` hint) in a statement block, then the optimizer automatically uses the cost-based approach.

> **See Also:** the "Using Hints" section on page 7-6 shows the syntax of each hint.

Specifying a Full Set of Hints

When using hints, in some cases, you may need to specify a full set of hints, so as to ensure the optimal execution plan. For example, if you have a very complex query, which consists of many table joins, and if you specify only the `INDEX` hint for a given table, then the optimizer needs to determine the remaining access paths to be used, as well as the corresponding join methods. Therefore, even though you gave the `INDEX` hint, the optimizer may not necessarily use that hint, because the

optimizer may have determined that the requested index cannot be used due to the join methods and access paths selected by the optimizer. In this particular example, we have specified the exact join order to be used, via the `ORDERED` hint, as well as the join methods to be used on the different tables.

```
SELECT /*+ ORDERED INDEX (b, jl_br_balances_n1) USE_NL (j b)
           USE_NL (glcc glf) USE_MERGE (gp gsb) */
 b.application_id ,
 b.set_of_books_id ,
 b.personnel_id,
 p.vendor_id Personnel,
 p.segment1 PersonnelNumber,
 p.vendor_name Name
FROM  jl_br_journals j,
      jl_br_balances b,
      gl_code_combinations glcc,
      fnd_flex_values_vl glf,
      gl_periods gp,
      gl_sets_of_books gsb,
      po_vendors p
WHERE  . . . . . . . . . . . .
```

Using Hints Against Views

By default, hints do not propagate inside a complex view. For example, if you specify a hint in your query that selects against a complex view, then that hint is not honored, because it is not pushed inside the view.

Note: If the view is a single-table, then the hint is not propagated.

Unless the hints are inside the base view, they may not be honored from a query against the view.

Local vs. Global Hints

Table hints (i.e., hints that specify a table) normally refer to tables in the `DELETE`, `SELECT`, or `UPDATE` statement in which the hint occurs, not to tables inside any views or subqueries referenced by the statement. When you want to specify hints for tables that appear inside views or subqueries, Oracle recommends using global hints instead of embedding the hint in the view or subquery. Any table hint described in this chapter can be transformed into a global hint by using an extended syntax for the table name.

> **See Also:** For information on how to create global hints, see "Global Hints" on page 7-37.

Using Hints

Hints for Optimization Approaches and Goals

The hints described in this section allow you to choose between the cost-based and the rule-based optimization approaches. With the cost-based approach, this also includes the goal of best throughput or best response time.

- ALL_ROWS
- FIRST_ROWS
- CHOOSE
- RULE

If a SQL statement has a hint specifying an optimization approach and goal, then the optimizer uses the specified approach regardless of the presence or absence of statistics, the value of the `OPTIMIZER_MODE` initialization parameter, and the `OPTIMIZER_MODE` parameter of the `ALTER SESSION` statement.

> **Note:** The optimizer goal applies only to queries submitted directly. Use hints to determine the access path for any SQL statements submitted from within PL/SQL. The `ALTER SESSION`... `SET OPTIMIZER_MODE` statement does not affect SQL that is run from within PL/SQL.

ALL_ROWS

The `ALL_ROWS` hint explicitly chooses the cost-based approach to optimize a statement block with a goal of best throughput (that is, minimum total resource consumption).

The syntax of this hint is as follows:

/*+ ALL_ROWS */

For example, the optimizer uses the cost-based approach to optimize this statement for best throughput:

```
SELECT /*+ ALL_ROWS */ empno, ename, sal, job
FROM emp
WHERE empno = 7566;
```

FIRST_ROWS

The `FIRST_ROWS` hint explicitly chooses the cost-based approach to optimize a statement block with a goal of best response time (minimum resource usage to return first row).

This hint causes the optimizer to make the following choices:

- If an index scan is available, then the optimizer may choose it over a full table scan.
- If an index scan is available, then the optimizer may choose a nested loops join over a sort-merge join whenever the associated table is the potential inner table of the nested loops.
- If an index scan is made available by an `ORDER BY` clause, then the optimizer may choose it to avoid a sort operation.

The syntax of this hint is as follows:

For example, the optimizer uses the cost-based approach to optimize this statement for best response time:

```
SELECT /*+ FIRST_ROWS */ empno, ename, sal, job
FROM emp
WHERE empno = 7566;
```

The optimizer ignores this hint in `DELETE` and `UPDATE` statement blocks and in `SELECT` statement blocks that contain any of the following syntax:

- Set operators (`UNION`, `INTERSECT`, `MINUS`, `UNION ALL`)
- `GROUP BY` clause
- `FOR UPDATE` clause
- Aggregate functions
- `DISTINCT` operator

These statements cannot be optimized for best response time, because Oracle must retrieve all rows accessed by the statement before returning the first row. If you

specify this hint in any of these statements, then the optimizer uses the cost-based approach and optimizes for best throughput.

If you specify either the `ALL_ROWS` or the `FIRST_ROWS` hint in a SQL statement, and if the data dictionary does not have statistics about tables accessed by the statement, then the optimizer uses default statistical values (such as allocated storage for such tables) to estimate the missing statistics and, subsequently, to choose an execution plan.

These estimates may not be as accurate as those gathered by the `DBMS_STATS` package. Therefore, use the `DBMS_STATS` package to gather statistics. If you specify hints for access paths or join operations along with either the `ALL_ROWS` or `FIRST_ROWS` hint, then the optimizer gives precedence to the access paths and join operations specified by the hints.

CHOOSE

The `CHOOSE` hint causes the optimizer to choose between the rule-based and cost-based approaches for a SQL statement. The optimizer bases its selection on the presence of statistics for the tables accessed by the statement. If the data dictionary has statistics for at least one of these tables, then the optimizer uses the cost-based approach and optimizes with the goal of best throughput. If the data dictionary does not have statistics for these tables, then it uses the rule-based approach.

The syntax of this hint is as follows:

→ /*+ → CHOOSE → */ →

Example

```
SELECT /*+ CHOOSE */ empno, ename, sal, job
FROM emp
WHERE empno = 7566;
```

RULE

The `RULE` hint explicitly chooses rule-based optimization for a statement block. It also makes the optimizer ignore other hints specified for the statement block. The syntax of this hint is as follows:

→ /*+ → RULE → */ →

Example The optimizer uses the rule-based approach for this statement:

```
SELECT --+ RULE
empno, ename, sal, job
FROM emp
WHERE empno = 7566;
```

The `RULE` hint, along with the rule-based approach, may not be supported in future releases of Oracle.

Hints for Access Methods

Each hint described in this section suggests an access method for a table.

- FULL
- ROWID
- CLUSTER
- HASH
- INDEX
- INDEX_ASC
- INDEX_COMBINE
- INDEX_JOIN
- INDEX_DESC
- INDEX_FFS
- NO_INDEX
- AND_EQUAL
- USE_CONCAT
- NO_EXPAND
- REWRITE
- NOREWRITE

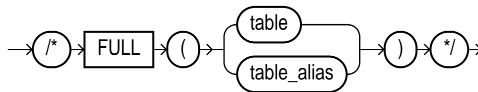
Specifying one of these hints causes the optimizer to choose the specified access path only if the access path is available based on the existence of an index or cluster and on the syntactic constructs of the SQL statement. If a hint specifies an unavailable access path, then the optimizer ignores it.

You must specify the table to be accessed exactly as it appears in the statement. If the statement uses an alias for the table, then use the alias rather than the table name in the hint. The table name within the hint should not include the schema name if the schema name is present in the statement.

Note: For access path hints, Oracle ignores the hint if you specify the `SAMPLE` option in the `FROM` clause of a `SELECT` statement. For more information on the `SAMPLE` option, see *Oracle8i Concepts* and *Oracle8i Reference.*

FULL

The `FULL` hint explicitly chooses a full table scan for the specified table. The syntax of this hint is as follows:



where `table` specifies the name or alias of the table on which the full table scan is to be performed. If your statement does not use aliases, then the table name is the default alias.

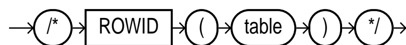
Example Oracle performs a full table scan on the `accounts` table to execute this statement, even if there is an index on the `accno` column that is made available by the condition in the `WHERE` clause:

```
SELECT /*+ FULL(A) don't use the index on accno */ accno, bal
FROM accounts a
WHERE accno = 7086854;
```

Note: Because the `accounts` table has alias "`a`", the hint must refer to the table by its alias rather than by its name. Also, do not specify schema names in the hint even if they are specified in the `FROM` clause.

ROWID

The ROWID hint explicitly chooses a table scan by rowid for the specified table. The syntax of the ROWID hint is:



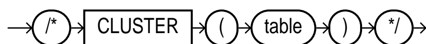
where table specifies the name or alias of the table on which the table access by rowid is to be performed.

Example

```
SELECT /*+ROWID(emp)*/ *
FROM emp
WHERE rowid > 'AAAAtkAABAAAFNTAAA' AND empno = 155;
```

CLUSTER

The CLUSTER hint explicitly chooses a cluster scan to access the specified table. It applies only to clustered objects. The syntax of the CLUSTER hint is:



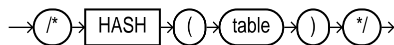
where table specifies the name or alias of the table to be accessed by a cluster scan.

Example

```
SELECT --+ CLUSTER
emp.ename, deptno
FROM emp, dept
WHERE deptno = 10
    AND emp.deptno = dept.deptno;
```

HASH

The HASH hint explicitly chooses a hash scan to access the specified table. It applies only to tables stored in a cluster. The syntax of the HASH hint is:

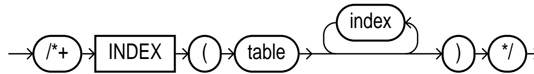


where table specifies the name or alias of the table to be accessed by a hash scan.

INDEX

The INDEX hint explicitly chooses an index scan for the specified table. You can use the INDEX hint for domain, B*-tree, and bitmap indexes. However, Oracle recommends using INDEX_COMBINE rather than INDEX for bitmap indexes, because it is a more versatile hint.

The syntax of the INDEX hint is:



where:

table	Specifies the name or alias of the table associated with the index to be scanned.
index	Specifies an index on which an index scan is to be performed.

This hint may optionally specify one or more indexes:

- If this hint specifies a single available index, then the optimizer performs a scan on this index. The optimizer does not consider a full table scan or a scan on another index on the table.
- If this hint specifies a list of available indexes, then the optimizer considers the cost of a scan on each index in the list and then performs the index scan with the lowest cost. The optimizer may also choose to scan multiple indexes from this list and merge the results, if such an access path has the lowest cost. The optimizer does not consider a full table scan or a scan on an index not listed in the hint.
- If this hint specifies no indexes, then the optimizer considers the cost of a scan on each available index on the table and then performs the index scan with the lowest cost. The optimizer may also choose to scan multiple indexes and merge the results, if such an access path has the lowest cost. The optimizer does not consider a full table scan.

For example, consider this query that selects the name, height, and weight of all male patients in a hospital:

```
SELECT name, height, weight
FROM patients
WHERE sex = 'm';
```

Assume that there is an index on the SEX column, and that this column contains the values m and f. If there are equal numbers of male and female patients in the hospital, then the query returns a relatively large percentage of the table's rows, and a full table scan is likely to be faster than an index scan. However, if a very small percentage of the hospital's patients are male, then the query returns a relatively small percentage of the table's rows, and an index scan is likely to be faster than a full table scan.

Barring the use of frequency histograms, the number of occurrences of each distinct column value is not available to the optimizer. The cost-based approach assumes that each value has an equal probability of appearing in each row. For a column having only two distinct values, the optimizer assumes each value appears in 50% of the rows, so the cost-based approach is likely to choose a full table scan rather than an index scan.

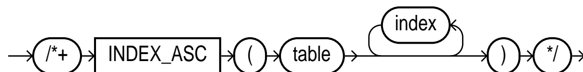
If you know that the value in the WHERE clause of your query appears in a very small percentage of the rows, then you can use the INDEX hint to force the optimizer to choose an index scan. In this statement, the INDEX hint explicitly chooses an index scan on the sex_index, the index on the sex column:

```
SELECT /*+ INDEX(patients sex_index) use sex_index because there are few
    male patients  */ name, height, weight
FROM patients
WHERE sex = 'm';
```

The INDEX hint applies to IN-list predicates; it forces the optimizer to use the hinted index, if possible, for an IN-list predicate. Multi-column IN-lists will not use an index.

INDEX_ASC

The INDEX_ASC hint explicitly chooses an index scan for the specified table. If the statement uses an index range scan, then Oracle scans the index entries in ascending order of their indexed values. The syntax of the INDEX_ASC hint is:

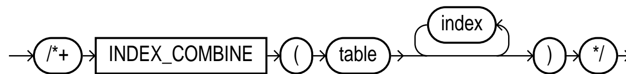


Each parameter serves the same purpose as in the INDEX hint.

Because Oracle's default behavior for a range scan is to scan index entries in ascending order of their indexed values, this hint does not specify anything more than the INDEX hint. However, you may want to use the INDEX_ASC hint to specify ascending range scans explicitly, should the default behavior change.

INDEX_COMBINE

The INDEX_COMBINE hint explicitly chooses a bitmap access path for the table. If no indexes are given as arguments for the INDEX_COMBINE hint, then the optimizer uses whatever Boolean combination of bitmap indexes has the best cost estimate for the table. If certain indexes are given as arguments, then the optimizer tries to use some Boolean combination of those particular bitmap indexes. The syntax of INDEX_COMBINE is:

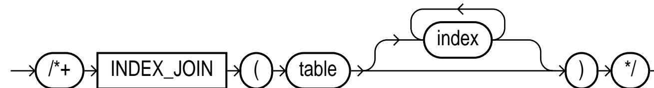


Example

```
SELECT /*+INDEX_COMBINE(emp sal_bmi hiredate_bmi)*/ *
FROM emp
WHERE sal < 50000 AND hiredate < '01-JAN-1990';
```

INDEX_JOIN

The INDEX_JOIN hint explicitly instructs the optimizer to use an index join as an access path. For the hint to have a positive effect, a sufficiently small number of indexes must exist that contain all the columns required to resolve the query.



where:

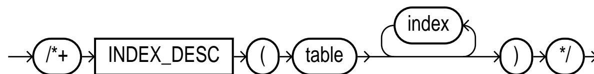
table	Specifies the name or alias of the table associated with the index to be scanned.
index	Specifies an index on which an index scan is to be performed.

Example

```
SELECT /*+INDEX_JOIN(emp sal_bmi hiredate_bmi)*/ sal, hiredate
FROM emp
WHERE sal < 50000;
```

INDEX_DESC

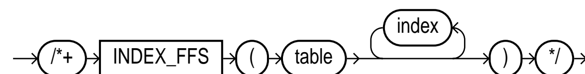
The INDEX_DESC hint explicitly chooses an index scan for the specified table. If the statement uses an index range scan, then Oracle scans the index entries in descending order of their indexed values. The syntax of the INDEX_DESC hint is:



Each parameter serves the same purpose as in the INDEX hint.

INDEX_FFS

This hint causes a fast full index scan to be performed rather than a full table scan. The syntax of INDEX_FFS is:



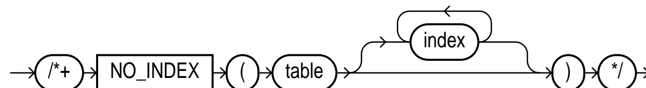
Example

```
SELECT /*+INDEX_FFS(emp emp_empno)*/ empno
FROM emp
WHERE empno > 200;
```

See Also: "Using Fast Full Index Scans" on page 12-8.

NO_INDEX

The NO_INDEX hint explicitly disallows a set of indexes for the specified table. The syntax of the NO_INDEX hint is:



- If this hint specifies a single available index, then the optimizer does not consider a scan on this index. Other indexes not specified are still considered.
- If this hint specifies a list of available indexes, then the optimizer does not consider a scan on any of the specified indexes. Other indexes not specified in the list are still considered.

- If this hint specifies no indexes, then the optimizer does not consider a scan on any index on the table. This behavior is the same as a NO_INDEX hint that specifies a list of all available indexes for the table.

The NO_INDEX hint applies to function-based, B*-tree, bitmap, cluster, or domain indexes.

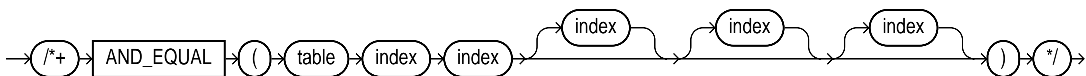
If a NO_INDEX hint and an index hint (INDEX, INDEX_ASC, INDEX_DESC, INDEX_COMBINE, or INDEX_FFS) both specify the same indexes, then both the NO_INDEX hint and the index hint are ignored for the specified indexes and the optimizer considers the specified indexes.

Example

```
SELECT /*+NO_INDEX(emp emp_empno)*/ empno
FROM emp
WHERE empno > 200;
```

AND_EQUAL

The AND_EQUAL hint explicitly chooses an execution plan that uses an access path that merges the scans on several single-column indexes. The syntax of the AND_EQUAL hint is:



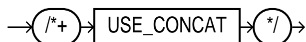
where:

table	Specifies the name or alias of the table associated with the indexes to be merged.
index	Specifies an index on which an index scan is to be performed. You must specify at least two indexes. You cannot specify more than five.

USE_CONCAT

The USE_CONCAT hint forces combined OR conditions in the WHERE clause of a query to be transformed into a compound query using the UNION ALL set operator. Normally, this transformation occurs only if the cost of the query using the concatenations is cheaper than the cost without them.

The USE_CONCAT hint turns off IN-list processing and OR-expands all disjunctions, including IN-lists. The syntax of this hint is:



Example

```
SELECT /*+USE_CONCAT*/ *
FROM emp
WHERE empno > 50 OR sal < 50000;
```

NO_EXPAND

The NO_EXPAND hint prevents the cost-based optimizer from considering OR-expansion for queries having OR conditions or IN-lists in the WHERE clause. Usually, the optimizer considers using OR expansion and uses this method if it decides the cost is lower than not using it. The syntax of this hint is:

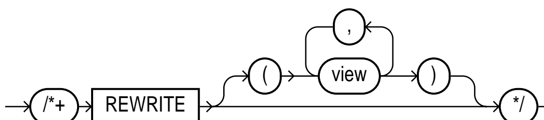
Example

```
SELECT /*+NO_EXPAND*/ *
FROM emp
WHERE empno = 50 OR empno = 100;
```

REWRITE

Use the REWRITE hint with or without a view list. If you use REWRITE with a view list and the list contains an eligible materialized view, then Oracle uses that view regardless of its cost. Oracle does not consider views outside of the list. If you do not specify a view list, then Oracle searches for an eligible materialized view and always uses it regardless of its cost.

The syntax of this hint is:



> **See Also:** For more information on materialized views, see *Oracle8i Concepts* and *Oracle8i Application Developer's Guide - Fundamentals.*

NOREWRITE

Use the NOREWRITE hint on any query block of a request. This hint disables query rewrite for the query block, overriding the setting of the parameter QUERY_REWRITE_ENABLED. The syntax of this hint is:

```
→(/*+)→[NOREWRITE]→(*/)→
```

Hints for Join Orders

The hints in this section suggest join orders:

- ORDERED
- STAR

ORDERED

The ORDERED hint causes Oracle to join tables in the order in which they appear in the FROM clause. The syntax of this hint is:

```
→(/*+)→[ORDERED]→(*/)→
```

For example, this statement joins table TAB1 to table TAB2 and then joins the result to table TAB3:

```
SELECT /*+ ORDERED */ tab1.col1, tab2.col2, tab3.col3
FROM tab1, tab2, tab3
WHERE tab1.col1 = tab2.col1
    AND tab2.col1 = tab3.col1;
```

If you omit the ORDERED hint from a SQL statement performing a join, then the optimizer chooses the order in which to join the tables. You may want to use the ORDERED hint to specify a join order if you know something about the number of rows selected from each table that the optimizer does not. Such information allows you to choose an inner and outer table better than the optimizer could.

STAR

The STAR hint forces a star query plan to be used, if possible. A star plan has the largest table in the query last in the join order and joins it with a nested loops join on a concatenated index. The STAR hint applies when there are at least three tables, the large table's concatenated index has at least three columns, and there are no conflicting access or join method hints. The optimizer also considers different permutations of the small tables.

The syntax of this hint is:

→(/*+)→ STAR →(*/)→

Usually, if you analyze the tables, then the optimizer selects 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) */
```

Where facts is the table and fact_concat is the index. A more general method is to use the STAR hint.

> **See Also:** *Oracle8i Concepts* for more information about star plans.

Hints for Join Operations

Each hint described in this section suggests a join operation for a table.

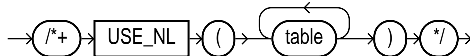
- USE_NL
- USE_MERGE
- USE_HASH
- DRIVING_SITE
- LEADING
- HASH_AJ and MERGE_AJ
- HASH_SJ and MERGE_SJ

You must specify a table to be joined exactly as it appears in the statement. If the statement uses an alias for the table, then you must use the alias rather than the table name in the hint. The table name within the hint should not include the schema name, if the schema name is present in the statement.

Use of the USE_NL and USE_MERGE hints is recommended with the ORDERED hint. Oracle uses these hints when the referenced table is forced to be the inner table of a join, and they are ignored if the referenced table is the outer table.

USE_NL

The USE_NL hint causes Oracle to join each specified table to another row source with a nested loops join using the specified table as the inner table. The syntax of the USE_NL hint is:



where table is the name or alias of a table to be used as the inner table of a nested loops join.

For example, consider this statement, which joins the accounts and customers tables. Assume that these tables are not stored together in a cluster:

```
SELECT accounts.balance, customers.last_name, customers.first_name
FROM accounts, customers
WHERE accounts.custno = customers.custno;
```

Because the default goal of the cost-based approach is best throughput, the optimizer chooses either a nested loops operation or a sort-merge operation to join these tables, depending on which is likely to return all the rows selected by the query more quickly.

However, you may want to optimize the statement for best response time, or the minimal elapsed time necessary to return the first row selected by the query, rather than best throughput. If so, then you can force the optimizer to choose a nested loops join by using the USE_NL hint. In this statement, the USE_NL hint explicitly chooses a nested loops join with the customers table as the inner table:

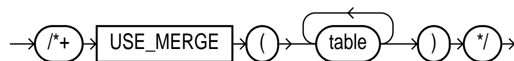
```
SELECT /*+ ORDERED USE_NL(customers) to get first row faster */
accounts.balance, customers.last_name, customers.first_name
FROM accounts, customers
WHERE accounts.custno = customers.custno;
```

In many cases, a nested loops join returns the first row faster than a sort-merge join. A nested loops join can return the first row after reading the first selected row from one table and the first matching row from the other and combining them, while a sort-merge join cannot return the first row until after reading and sorting all

selected rows of both tables and then combining the first rows of each sorted row source.

USE_MERGE

The `USE_MERGE` hint causes Oracle to join each specified table with another row source with a sort-merge join. The syntax of the `USE_MERGE` hint is:



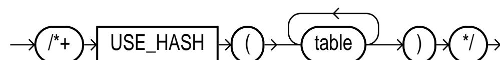
where `table` is a table to be joined to the row source resulting from joining the previous tables in the join order using a sort-merge join.

Example

```
SELECT /*+USE_MERGE(emp dept)*/ *
FROM emp, dept
WHERE emp.deptno = dept.deptno;
```

USE_HASH

The `USE_HASH` hint causes Oracle to join each specified table with another row source with a hash join. The syntax of the `USE_HASH` hint is:



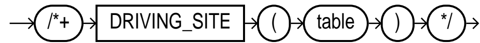
where `table` is a table to be joined to the row source resulting from joining the previous tables in the join order using a hash join.

Example

```
SELECT /*+use_hash(emp dept)*/ *
FROM emp, dept
WHERE emp.deptno = dept.deptno;
```

DRIVING_SITE

The DRIVING_SITE hint forces query execution to be done at a different site than that selected by Oracle. This hint can be used with either rule-based or cost-based optimization. The syntax of this hint is:



where table is the name or alias for the table at which site the execution should take place.

Example

```
SELECT /*+DRIVING_SITE(dept)*/ *
FROM emp, dept@rsite
WHERE emp.deptno = dept.deptno;
```

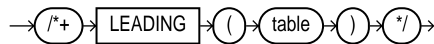
If this query is executed without the hint, then rows from dept are sent to the local site, and the join is executed there. With the hint, the rows from emp are sent to the remote site, and the query is executed there, returning the result to the local site.

This hint is useful if you are using distributed query optimization.

> **See Also:** *Oracle8i Distributed Database Systems*

LEADING

The LEADING hint causes Oracle to use the specified table as the first table in the join order. The syntax of the hint is:



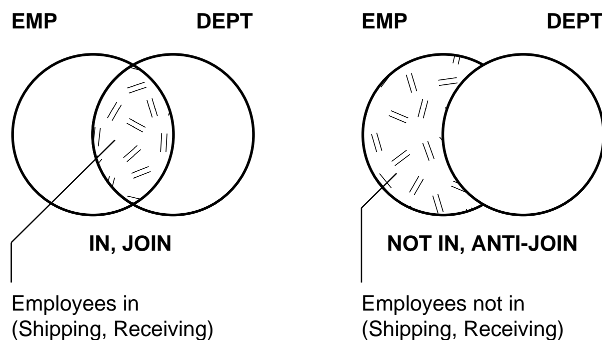
Where *table* is the name or alias of a table to be used as the first table in the join order.

If you specify two or more LEADING hints on different tables, then all of them are ignored. If you specify the ORDERED hint, then it overrides all LEADING hints.

HASH_AJ and MERGE_AJ

As illustrated in Figure 7–1, the SQL IN predicate can be evaluated using a join to intersect two sets. Thus emp.deptno can be joined to dept.deptno to yield a list of employees in a set of departments.

Figure 7–1 Parallel Hash Anti-join



Alternatively, the SQL NOT IN predicate can be evaluated using an anti-join to subtract two sets. Thus emp.deptno can be anti-joined to dept.deptno to select all employees who are not in a set of departments, and you can get a list of all employees who are not in the Shipping or Receiving departments.

For a specific query, place the MERGE_AJ or HASH_AJ hints into the NOT IN subquery. MERGE_AJ uses a sort-merge anti-join and HASH_AJ uses a hash anti-join.

For example:

```
SELECT * FROM emp
WHERE ename LIKE 'J%'
    AND deptno IS NOT NULL
    AND deptno NOT IN (SELECT /*+ HASH_AJ */ deptno
        FROM dept
        WHERE deptno IS NOT NULL
        AND loc = 'DALLAS');
```

If you want the anti-join transformation always to occur if the conditions in the previous section are met, then set the ALWAYS_ANTI_JOIN initialization parameter to MERGE or HASH. The transformation to the corresponding anti-join type then takes place whenever possible.

HASH_SJ and MERGE_SJ

For a specific query, place the HASH_SJ or MERGE_SJ hint into the EXISTS subquery. HASH_SJ uses a hash semi-join and MERGE_SJ uses a sort merge semi-join. For example:

```
SELECT * FROM dept
WHERE exists (SELECT /*+HASH_SJ*/ *
  FROM emp
  WHERE emp.deptno = dept.deptno
        AND sal > 200000);
```

This converts the subquery into a special type of join between t1 and t2 that preserves the semantics of the subquery. That is, even if there is more than one matching row in `t2` for a row in `t1`, the row in `t1` is returned only once.

A subquery is evaluated as a semi-join only with these limitations:

- There can only be one table in the subquery.
- The outer query block must not itself be a subquery.
- The subquery must be correlated with an equality predicate.
- The subquery must have no `GROUP BY`, `CONNECT BY`, or `ROWNUM` references.

If you want the semi-join transformation always to occur if the conditions in the previous section are met, then set the `ALWAYS_SEMI_JOIN` initialization parameter to `HASH` or `MERGE`. The transformation to the corresponding semi-join type then takes place whenever possible.

Hints for Parallel Execution

The hints described in this section determine how statements are parallelized or not parallelized when using parallel execution.

- PARALLEL
- NOPARALLEL
- PQ_DISTRIBUTE
- APPEND
- NOAPPEND
- PARALLEL_INDEX
- NOPARALLEL_INDEX

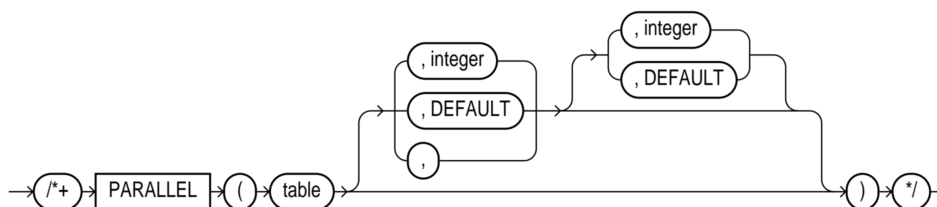
> **See Also:** For more information on parallel execution, see *Oracle8i Data Warehousing Guide*.

PARALLEL

The PARALLEL hint lets you specify the desired number of concurrent servers that can be used for a parallel operation. The hint applies to the INSERT, UPDATE, and DELETE portions of a statement as well as to the table scan portion.

> **Note:** The number of servers that can be used is twice the value in the PARALLEL hint if sorting or grouping operations also take place.

If any parallel restrictions are violated, then the hint is ignored. The syntax is:



The PARALLEL hint must use the table alias if an alias is specified in the query. The hint can then take two values separated by commas after the table name. The first value specifies the degree of parallelism for the given table, the second value specifies how the table is to be split among the instances of a parallel server. Specifying DEFAULT or no value signifies that the query coordinator should examine the settings of the initialization parameters (described in a later section) to determine the default degree of parallelism.

In the following example, the PARALLEL hint overrides the degree of parallelism specified in the emp table definition:

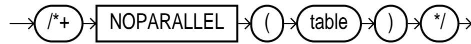
```
SELECT /*+ FULL(scott_emp) PARALLEL(scott_emp, 5) */ ename
FROM scott.emp scott_emp;
```

In the next example, the PARALLEL hint overrides the degree of parallelism specified in the emp table definition and tells the optimizer to use the default degree of parallelism determined by the initialization parameters. This hint also specifies that the table should be split among all of the available instances, with the default degree of parallelism on each instance.

```
SELECT /*+ FULL(scott_emp) PARALLEL(scott_emp, DEFAULT,DEFAULT) */ ename
FROM scott.emp scott_emp;
```

NOPARALLEL

You can use the NOPARALLEL hint to override a PARALLEL specification in the table clause. In general, hints take precedence over table clauses. The syntax of this hint is:



The following example illustrates the NOPARALLEL hint:

```
SELECT /*+ NOPARALLEL(scott_emp) */ ename
FROM scott.emp scott_emp;
```

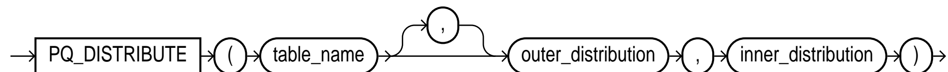
PQ_DISTRIBUTE

Use the PQ_DISTRIBUTE hint to improve parallel join operation performance. Do this by specifying how rows of joined tables should be distributed among producer and consumer query servers. Using this hint overrides decisions the optimizer would normally make.

Use the EXPLAIN PLAN statement to identify the distribution chosen by the optimizer. The optimizer ignores the distribution hint if both tables are serial.

> **See Also:** For more information on how Oracle parallelizes join operations, see *Oracle8i Concepts*.

The syntax of this hint is:



where:

table_name	Name or alias of a table to be used as the inner table of a join.
outer_distribution	The distribution for the outer table.
inner_distribution	The distribution for the inner table.

There are six combinations for table distribution. Only a subset of distribution method combinations for the joined tables is valid, as explained in Table 7–1.

Table 7–1 Distribution Hint Combinations

Distribution	Interpretation
Hash, Hash	Maps the rows of each table to consumer query servers using a hash function on the join keys. When mapping is complete, each query server performs the join between a pair of resulting partitions. This hint is recommended when the tables are comparable in size and the join operation is implemented by hash-join or sort-merge join.
Broadcast, None	All rows of the outer table are broadcast to each query server. The inner table rows are randomly partitioned. This hint is recommended when the outer table is very small compared to the inner table. A rule-of-thumb is: *Use the Broadcast/None hint if the size of the inner table * number of query servers > size of the outer table.*
None, Broadcast	All rows of the inner table are broadcast to each consumer query server. The outer table rows are randomly partitioned. This hint is recommended when the inner table is very small compared to the outer table. A rule-of-thumb is: *Use the None/Broadcast hint if the size of the inner table * number of query servers < size of the outer table.*
Partition, None	Maps the rows of the outer table using the partitioning of the inner table. The inner table must be partitioned on the join keys. This hint is recommended when the number of partitions of the outer table is equal to or nearly equal to a multiple of the number of query servers, for example, 14 partitions and 15 query servers. **Note:** The optimizer ignores this hint if the inner table is not partitioned or not equijoined on the partitioning key.
None, Partition	Maps the rows of the inner table using the partitioning of the outer table. The outer table must be partitioned on the join keys. This hint is recommended when the number of partitions of the outer table is equal to or nearly equal to a multiple of the number of query servers, for example, 14 partitions and 15 query servers. **Note:** The optimizer ignores this hint if the outer table is not partitioned or not equijoined on the partitioning key.
None, None	Each query server performs the join operation between a pair of matching partitions, one from each table. Both tables must be equi-partitioned on the join keys.

Examples Given two tables, R and S, that are joined using a hash-join, the following query contains a hint to use hash distribution:

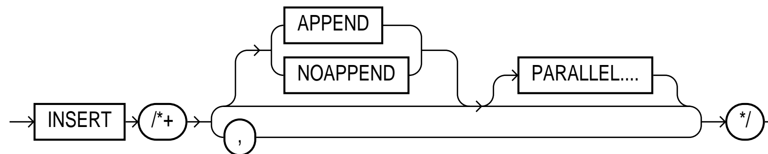
```
SELECT <column_list> /*+ORDERED PQ_DISTRIBUTE(s HASH, HASH) USE_HASH (s)*/
FROM r,s
WHERE r.c=s.c;
```

To broadcast the outer table r, the query should be:

```
SELECT <column list> /*+ORDERED PQ_DISTRIBUTE(s BROADCAST, NONE) USE_HASH (s) */
FROM r,s
WHERE r.c=s.c;
```

APPEND

When you use the APPEND hint for INSERT, data is simply appended to a table. Existing free space in the blocks currently allocated to the table is not used. The syntax of this hint is:



If INSERT is parallelized using the PARALLEL hint or clause, then append mode is used by default. You can use NOAPPEND to override append mode. The APPEND hint applies to both serial and parallel insert.

The append operation is performed in LOGGING or NOLOGGING mode, depending on whether the [NO] option is set for the table in question. Use the ALTER TABLE... [NO]LOGGING statement to set the appropriate value.

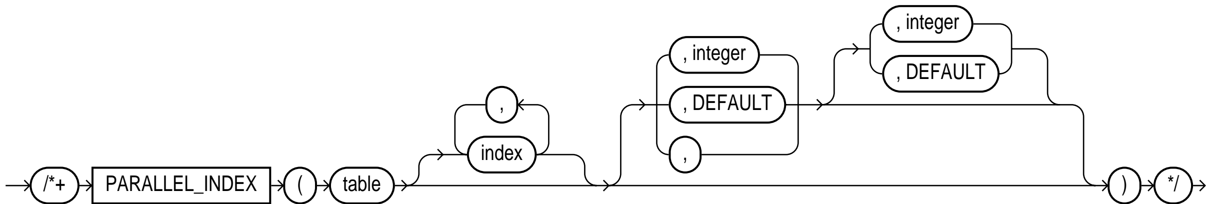
Note: Certain restrictions apply to the APPEND hint; these are detailed in *Oracle8i Concepts*. If any of these restrictions are violated, then the hint is ignored.

NOAPPEND

Use NOAPPEND to override append mode.

PARALLEL_INDEX

Use the PARALLEL_INDEX hint to specify the desired number of concurrent servers that can be used to parallelize index range scans for partitioned indexes. The syntax of the PARALLEL_INDEX hint is:



where:

table	Specifies the name or alias of the table associated with the index to be scanned.
index	Specifies an index on which an index scan is to be performed (optional).

The hint can take two values separated by commas after the table name. The first value specifies the degree of parallelism for the given table. The second value specifies how the table is to be split among the instances of a parallel server. Specifying DEFAULT or no value signifies the query coordinator should examine the settings of the initialization parameters (described in a later section) to determine the default degree of parallelism.

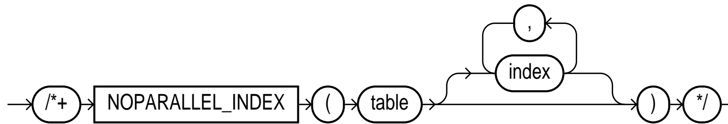
Example

```
SELECT /*+ PARALLEL_INDEX(table1, index1, 3, 2) +/
```

In this example, there are 3 parallel execution processes to be used on each of 2 instances.

NOPARALLEL_INDEX

Use the NOPARALLEL_INDEX hint to override a PARALLEL attribute setting on an index. In this way you can avoid a parallel index scan operation. The syntax of this hint is:



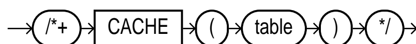
Additional Hints

Several additional hints are included in this section:

- CACHE
- NOCACHE
- MERGE
- NO_MERGE
- UNNEST
- NO_UNNEST
- PUSH_PRED
- NO_PUSH_PRED
- PUSH_SUBQ
- STAR_TRANSFORMATION
- ORDERED_PREDICATES

CACHE

The CACHE hint specifies that the blocks retrieved for this table are placed at the most recently used end of the LRU list in the buffer cache when a full table scan is performed. This option is useful for small lookup tables. The syntax of this hint is:

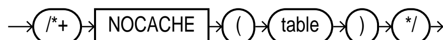


In the following example, the CACHE hint overrides the table's default caching specification:

```
SELECT /*+ FULL (scott_emp) CACHE(scott_emp) */ ename
FROM scott.emp scott_emp;
```

NOCACHE

The `NOCACHE` hint specifies that the blocks retrieved for this table are placed at the least recently used end of the LRU list in the buffer cache when a full table scan is performed. This is the normal behavior of blocks in the buffer cache. The syntax of this hint is:



Example

```
SELECT /*+ FULL(scott_emp) NOCACHE(scott_emp) */ ename
FROM scott.emp scott_emp;
```

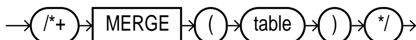
Note: The `CACHE` and `NOCACHE` hints affect system statistics "table scans(long tables)" and "table scans(short tables)", as shown in the `V$SYSSTAT` view.

MERGE

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.

Complex merging is not cost-based--that is, the accessing query block must include the `MERGE` hint. Without this hint, the optimizer uses another approach.

Merge a view on a per-query basis by using the `MERGE` hint. The syntax of this hint is:



Example

```
SELECT /*+MERGE(v)*/ e1.ename, e1.sal, v.avg_sal
FROM emp e1,
  (SELECT deptno, avg(sal) avg_sal
   FROM emp e2
```

```
   GROUP BY deptno) v
WHERE e1.deptno = v.deptno AND e1.sal > v.avg_sal;
```

Note: This example requires complex view merging to be enabled.

NO_MERGE

The NO_MERGE hint causes Oracle not to merge mergeable views. The syntax of the NO_MERGE hint is:

```
→ /*+ → NO_MERGE → ( → table → ) → */ →
```

This hint lets the user have more influence over the way in which the view is accessed.

Example

```
SELECT /*+NO_MERGE(dallasdept)*/ e1.ename, dallasdept.dname
FROM emp e1,
  (SELECT deptno, dname
   FROM dept
   WHERE loc = 'DALLAS') dallasdept
WHERE e1.deptno = dallasdept.deptno;
```

This causes view v not to be merged.

When the NO_MERGE hint is used without an argument, it should be placed in the view query block. When NO_MERGE is used with the view name as an argument, it should be placed in the surrounding query.

UNNEST

Setting the UNNEST_SUBQUERY session parameter to TRUE enables subquery unnesting. Subquery unnesting unnests and merges the body of the subquery into the body of the statement that contains it, allowing the optimizer to consider them together when evaluating access paths and joins.

UNNEST_SUBQUERY first verifies if the statement is valid. If the statement is not valid, then subquery unnesting cannot proceed. The statement must then must pass a heuristic test.

> **See Also:** For more information on unnesting nested subqueries and the conditions that make a subquery block valid, see the *Oracle8i SQL Reference*. For more information on the UNNEST_SUBQUERY parameter and managing views, see Chapter 9, "Optimizing SQL Statements".

The UNNEST hint checks the subquery block for validity only. If it is valid, then subquery unnesting is enabled without Oracle checking the heuristics.

NO_UNNEST

If you enabled subquery unnesting with the UNNEST_SUBQUERY parameter, then the NO_UNNEST hint turns it off for specific subquery blocks.

> **Note:** The hints HASH_SJ, HASH_AJ, MERGE_SJ, and MERGE_AJ take precedence over this hint.

Example

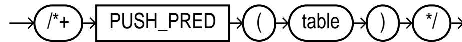
The following examples show situations where it might not be optimal to enable subquery unnesting.

```
SELECT *
FROM t_4k, t_5k
WHERE t_5k.ten = t_4k.thousand AND t_4k.thousand < 10
    AND t_5k.unique3 < 10
    AND t_5k.thousand < ALL (SELECT /*+ NO_UNNEST */ z_4k.thousand
        FROM z_4k
        WHERE z_4k.ten < t_4k.hundred);

SELECT SUM(l_extendedprice)
FROM lineitem, parts
WHERE p_partkey = l_partkey and p_brand = 'Brand#23'
    AND p_container = 'MED BOX'
    AND l_quantity < (SELECT AVG (l_quantity)
        FROM lineitem
        WHERE l_partkey = p_partkey);
```

PUSH_PRED

Use the PUSH_PRED hint to force pushing of a join predicate into the view. The syntax of this hint is:



Example

```
SELECT /*+ PUSH_PRED(v) */ t1.x, v.y
FROM t1
    (SELECT t2.x, t3.y
    FROM t2, t3
    WHERE t2.x = t3.x) v
WHERE t1.x = v.x and t1.y = 1;
```

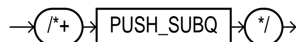
NO_PUSH_PRED

Use the NO_PUSH_PRED hint to prevent pushing of a join predicate into the view. The syntax of this hint is:

PUSH_SUBQ

The PUSH_SUBQ hint causes non-merged subqueries to be evaluated at the earliest possible place in the execution plan. Normally, subqueries that are not merged are executed as the last step in the execution plan. If the subquery is relatively inexpensive and reduces the number of rows significantly, then it improves performance to evaluate the subquery earlier.

The hint has no effect if the subquery is applied to a remote table or one that is joined using a merge join. The syntax of this hint is:



STAR_TRANSFORMATION

The STAR_TRANSFORMATION hint makes the optimizer use the best plan in which the transformation has been used. Without the hint, the optimizer could make a cost-based decision to use the best plan generated without the transformation, instead of the best plan for the transformed query.

Even if the hint is given, there is no guarantee that the transformation will take place. The optimizer only generates the subqueries if it seems reasonable to do so. If no subqueries are generated, then there is no transformed query, and the best plan for the untransformed query is used, regardless of the hint.

The syntax of this hint is:

```
—> (/*+) —> [STAR_TRANSFORMATION] —> (*/) —>
```

> **See Also:** *Oracle8i Concepts* has a full discussion of star transformation. Also, the *Oracle8i Reference* describes `STAR_TRANSFORMATION_ENABLED`; this parameter causes the optimizer to consider performing a star transformation.

ORDERED_PREDICATES

The `ORDERED_PREDICATES` hint forces the optimizer to preserve the order of predicate evaluation, except for predicates used as index keys. Use this hint in the `WHERE` clause of `SELECT` statements.

If you do not use the `ORDERED_PREDICATES` hint, then Oracle evaluates all predicates in the order specified by the following rules. Predicates:

- Without user-defined functions, type methods, or subqueries are evaluated first, in the order specified in the `WHERE` clause.
- With user-defined functions and type methods that have user-computed costs are evaluated next, in increasing order of their cost.
- With user-defined functions and type methods without user-computed costs are evaluated next, in the order specified in the `WHERE` clause.
- Not specified in the `WHERE` clause (for example, predicates transitively generated by the optimizer) are evaluated next.
- With subqueries are evaluated last in the order specified in the `WHERE` clause.

> **Note:** As mentioned, you cannot use the `ORDERED_PREDICATES` hint to preserve the order of predicate evaluation on index keys.

The syntax of this hint is:

```
→(/*+)→[ORDERED_PREDICATES]→(*/)→
```

See Also: *Oracle8i Concepts*

Using Hints with Views

Oracle does not encourage you to use hints inside or on views (or subqueries). This is because you can define views in one context and use them in another. However, such hints can result in unexpected plans. In particular, hints inside views or on views are handled differently depending on whether the view is mergeable into the top-level query.

Should you decide, nonetheless, to use hints with views, the following sections describe the behavior in each case.

- Hints and Mergeable Views
- Hints and Nonmergeable Views

If you want to specify a hint for a table in a view or subquery, then the global hint syntax is recommended. The following section describes this in detail.

- Global Hints

Hints and Mergeable Views

This section describes hint behavior with mergeable views.

Optimization Approaches and Goal Hints Optimization approach and goal hints can occur in a top-level query or inside views.

- If there is such a hint in the top-level query, then that hint is used regardless of any such hints inside the views.
- If there is no top-level optimizer mode hint, then mode hints in referenced views are used as long as all mode hints in the views are consistent.
- If two or more mode hints in the referenced views conflict, then all mode hints in the views are discarded and the session mode is used, whether default or user-specified.

Access Method and Join Hints on Views Access method and join hints on referenced views are ignored unless the view contains a single table (or references another view with a single table). For such single-table views, an access method hint or a join hint on the view applies to the table inside the view.

Access Method and Join Hints Inside Views Access method and join hints can appear in a view definition.

- If the view is a subquery (that is, if it appears in the `FROM` clause of a `SELECT` statement), then all access method and join hints inside the view are preserved when the view is merged with the top-level query.
- For views that are not subqueries, access method and join hints in the view are preserved only if the top-level query references no other tables or views (that is, if the `FROM` clause of the `SELECT` statement contains only the view).

Parallel Execution Hints on Views `PARALLEL`, `NOPARALLEL`, `PARALLEL_INDEX` and `NOPARALLEL_INDEX` hints on views are always recursively applied to all the tables in the referenced view. Parallel execution hints in a top-level query override such hints inside a referenced view.

Parallel Execution Hints Inside Views `PARALLEL`, `NOPARALLEL`, `PARALLEL_INDEX` and `NOPARALLEL_INDEX` hints inside views are preserved when the view is merged with the top-level query. Parallel execution hints on the view in a top-level query override such hints inside a referenced view.

Hints and Nonmergeable Views

With non-mergeable views, optimization approach and goal hints inside the view are ignored: the top-level query decides the optimization mode.

Because non-mergeable views are optimized separately from the top-level query, access method and join hints inside the view are always preserved. For the same reason, access method hints on the view in the top-level query are ignored.

However, join hints on the view in the top-level query are preserved because, in this case, a non-mergeable view is similar to a table.

Global Hints

Table hints (i.e., hints that specify a table) normally refer to tables in the `DELETE`, `SELECT`, or `UPDATE` statement in which the hint occurs, not to tables inside any views or subqueries referenced by the statement. When you want to specify hints for tables that appear inside views or subqueries, you should use global hints

instead of embedding the hint in the view or subquery. You can transform any table hint in this chapter into a global hint by using an extended syntax for the table name, as described below.

Consider the following view definitions and SELECT statement:

```
CREATE VIEW v1 AS
    SELECT *
    FROM emp
    WHERE empno < 100;

CREATE VIEW v2 AS
    SELECT v1.empno empno, dept.deptno deptno
    FROM v1, dept
    WHERE v1.deptno = dept.deptno;

SELECT /*+ INDEX(v2.v1.emp emp_empno) FULL(v2.dept) */ *
    FROM v2
    WHERE deptno = 20;
```

The view V1 retrieves all employees whose employee number is less than 100. The view V2 performs a join between the view V1 and the department table. The SELECT statement retrieves rows from the view V2 restricting it to the department whose number is 20.

There are two global hints in the SELECT statement. The first hint specifies an index scan for the employee table referenced in the view V1, which is referenced in the view V2. The second hint specifies a full table scan for the department table referenced in the view V2. Note the dotted syntax for the view tables.

A hint such as:

```
INDEX(emp emp_empno)
```

in the SELECT statement is ignored because the employee table does not appear in the FROM clause of the SELECT statement.

The global hint syntax also applies to unmergeable views. Consider the following SELECT statement:

```
SELECT /*+ NO_MERGE(v2) INDEX(v2.v1.emp emp_empno) FULL(v2.dept) */ *
FROM v2
WHERE deptno = 20;
```

It causes V2 not to be merged, and specifies access path hints for the employee and department tables. These hints are pushed down into the (nonmerged) view V2.

If a global hint references a UNION or UNION ALL view, then the hint is applied to the first branch that contains the hinted table. Consider the INDEX hint in the following SELECT statement:

```
SELECT /*+ INDEX(v.emp emp_empno) */ *
FROM (SELECT *
    FROM emp
    WHERE empno < 50
    UNION ALL
    SELECT *
    FROM emp
    WHERE empno > 1000) v
WHERE deptno = 20;
```

The INDEX hint applies to the employee table in the first branch of the UNION ALL view v, not to the employee table in the second branch.

8

Gathering Statistics

This chapter explains why statistics are important for the cost-based optimizer, and how to gather and use statistics.

This chapter contains the following sections:

- Understanding Statistics
- Generating Statistics
- Using Statistics
- Using Histograms

Understanding Statistics

The cost-based optimization approach uses statistics to calculate the selectivity of predicates and to estimate the cost of each execution plan. *Selectivity* is the fraction of rows in a table that the SQL statement's predicate chooses. The optimizer uses the selectivity of a predicate to estimate the cost of a particular access method and to determine the optimal join order.

Statistics quantify the data distribution and storage characteristics of tables, columns, indexes, and partitions. The optimizer uses these statistics to estimate how much I/O and memory are required to execute a SQL statement using a particular execution plan. The statistics are stored in the data dictionary, and they can be exported from one database and imported into another (for example, to transfer production statistics to a test system to simulate the real environment, even though the test system may only have small samples of data).

You must gather statistics on a regular basis to provide the optimizer with information about schema objects. New statistics should be gathered after a schema object's data or structure are modified in ways that make the previous statistics inaccurate. For example, after loading a significant number of rows into a table, you should collect new statistics on the number of rows. After updating data in a table, you do not need to collect new statistics on the number of rows but you might need new statistics on the average row length.

Statistics can be generated with the `ANALYZE` statement or with the package `DBMS_STATS`.

The statistics generated include the following:

- Table statistics
  - Number of rows
  - Number of blocks
  - Number of empty blocks
  - Average row length
- Column statistics
  - Number of distinct values (NDV) in column
  - Number of nulls in column
  - Data distribution (histogram)
- Index statistics

- Number of leaf blocks
- Levels
- Clustering factor

Generating Statistics

Because the cost-based approach relies on statistics, you should generate statistics for all tables and clusters and all types of indexes accessed by your SQL statements before using the cost-based approach. If the size and data distribution of your tables change frequently, then you should generate these statistics regularly to ensure the statistics accurately represent the data in the tables.

Oracle generates statistics using the following techniques:

- Estimation based on random data sampling
- Exact computation
- User-defined statistics collection methods

To perform an exact computation, Oracle requires enough space to perform a scan and sort of the table. If there is not enough space in memory, then temporary space may be required. For estimations, Oracle requires enough space to perform a scan and sort of only the rows in the requested sample of the table. For indexes, computation does not take up as much time or space, so it is best to perform a full computation.

Some statistics are always computed exactly, such as the number of data blocks currently containing data in a table or the depth of an index from its root block to its leaf blocks.

Use estimation for tables and clusters rather than computation, unless you need exact values. Because estimation rarely sorts, it is often much faster than computation, especially for large tables.

To estimate statistics, Oracle selects a random sample of data. You can specify the sampling percentage and whether sampling should be based on rows or blocks.

- *Row sampling* reads rows without regard to their physical placement on disk. This provides the most random data for estimates, but it can result in reading more data than necessary. For example, in the worst case a row sample might select one row from each block, requiring a full scan of the table or index.
- *Block sampling* reads a random sample of blocks and uses all of the rows in those blocks for estimates. This reduces the amount of I/O activity for a given sample

size, but it can reduce the randomness of the sample if rows are not randomly distributed on disk. Block sampling is not available for index statistics.

When you generate statistics for a table, column, or index, if the data dictionary already contains statistics for the object, then Oracle updates the existing statistics. Oracle also invalidates any currently parsed SQL statements that access the object.

The next time such a statement executes, the optimizer automatically chooses a new execution plan based on the new statistics. Distributed statements issued on remote databases that access the analyzed objects use the new statistics the next time Oracle parses them.

When you associate a statistics type with a column or domain index, Oracle calls the statistics collection method in the statistics type if you analyze the column or domain index.

Statistics for Partitioned Schema Objects

Partitioned schema objects may contain multiple sets of statistics. They can have statistics which refer to the entire schema object as a whole (global statistics), they can have statistics which refer to an individual partition, and they can have statistics which refer to an individual subpartition of a composite partitioned object.

Unless the query predicate narrows the query to a single partition, the optimizer uses the global statistics. Because most queries are not likely to be this restrictive, it is most important to have accurate global statistics. Intuitively, it may seem that generating global statistics from partition-level statistics should be straightforward; however, this is only true for some of the statistics. For example, it is very difficult to figure out the number of distinct values for a column from the number of distinct values found in each partition because of the possible overlap in values. Therefore, actually gathering global statistics with the `DBMS_STATS` package is highly recommended, rather than calculating them with the `ANALYZE` statement.

Note: Oracle currently does not gather global histogram statistics.

Using the ANALYZE Statement

The `ANALYZE` statement can generate statistics for cost-based optimization. However, using `ANALYZE` for this purpose is not recommended because of various restrictions, for example:

- `ANALYZE` always runs serially.

- ANALYZE calculates global statistics for partitioned tables and indexes instead of gathering them directly. This can lead to inaccuracies for some statistics, such as the number of distinct values.
  - For partitioned tables and indexes, ANALYZE gathers statistics for the individual partitions and then calculates the global statistics from the partition statistics.
  - For composite partitioning, ANALYZE gathers statistics for the subpartitions and then calculates the partition statistics and global statistics from the subpartition statistics.
- ANALYZE cannot overwrite or delete some of the values of statistics that were gathered by DBMS_STATS.

ANALYZE can gather additional information that is not used by the optimizer, such as information about chained rows and the structural integrity of indexes, tables, and clusters. DBMS_STATS does not gather this information.

> **See Also:** For detailed information about the ANALYZE statement, see *Oracle8i SQL Reference*.

Using the DBMS_STATS Package

The PL/SQL package DBMS_STATS lets you generate and manage statistics for cost-based optimization. You can use this package to gather, modify, view, and delete statistics. You can also use this package to store sets of statistics.

The DBMS_STATS package can gather statistics on indexes, tables, columns, and partitions, as well as statistics on all schema objects in a schema or database. It does not gather cluster statistics—you can use DBMS_STATS to gather statistics on the individual tables instead of the whole cluster.

The statistics-gathering operations can run either serially or in parallel. Whenever possible, DBMS_STATS calls a parallel query to gather statistics with the specified degree of parallelism; otherwise, it calls a serial query or the ANALYZE statement. Index statistics are not gathered in parallel.

For partitioned tables and indexes, DBMS_STATS can gather separate statistics for each partition as well as global statistics for the entire table or index. Similarly, for composite partitioning DBMS_STATS can gather separate statistics for subpartitions, partitions, and the entire table or index. Depending on the SQL statement being optimized, the optimizer may choose to use either the partition (or subpartition) statistics or the global statistics.

DBMS_STATS gathers statistics only for cost-based optimization; it does not gather other statistics. For example, the table statistics gathered by DBMS_STATS include the number of rows, number of blocks currently containing data, and average row length but not the number of chained rows, average free space, or number of unused data blocks.

Note: Currently, the DBMS_STATS package does not call statistics collection methods associated with individual columns. Use the ANALYZE statement to gather such information.

See Also: For more information about the DBMS_STATS package, see *Oracle8i Supplied PL/SQL Packages Reference.* For more information about user-defined statistics, see the *Oracle8i Data Cartridge Developer's Guide.*

Gathering Statistics with the DBMS_STATS Package

Table 8–1 lists the procedures in the DBMS_STATS package for gathering statistics:

Table 8–1 Statistics Gathering Procedures in the DBMS_STATS Package

Procedure	Description
GATHER_INDEX_STATS	Collects index statistics.
GATHER_TABLE_STATS	Collects table, column, and index statistics.
GATHER_SCHEMA_STATS	Collects statistics for all objects in a schema.
GATHER_DATABASE_STATS	Collects statistics for all objects in a database.

See Also: For syntax and examples of all DBMS_STATS procedures, see *Oracle8i Supplied PL/SQL Packages Reference.*

Gathering Index Statistics

Oracle can gather some statistics automatically while creating or rebuilding a B*-tree or bitmap index. The COMPUTE STATISTICS option of CREATE INDEX or ALTER INDEX ... REBUILD enables this gathering of statistics.

Note: COMPUTE STATISTICS always gathers exact statistics.

The statistics that Oracle gathers for the COMPUTE STATISTICS option depend on whether the index is partitioned or nonpartitioned.

- For a nonpartitioned index, Oracle gathers index, table, and column statistics while creating or rebuilding the index. In a concatenated-key index, the column statistics refer only to the leading column of the key.
- For a partitioned index, Oracle does not gather any table or column statistics while creating the index or rebuilding its partitions.
  - While creating a partitioned index, Oracle gathers index statistics for each partition and for the entire index. If the index uses composite partitioning, then Oracle also gathers statistics for each subpartition.
  - While rebuilding a partition or subpartition of an index, Oracle gathers index statistics only for that partition or subpartition.

To ensure correctness of the statistics Oracle always uses base tables when creating an index with the COMPUTE STATISTICS option, even if another index is available that could be used to create the index.

If you do not use the COMPUTE STATISTICS clause, or if you have made major DML changes, then use the DBMS_STATS.GATHER_INDEX_STATS procedure to collect index statistics. The GATHER_INDEX_STATS procedure does not run in parallel.

Using this procedure is equivalent to running the following:

```
ANALYZE INDEX [ownname.]indname [PARTITION partname] COMPUTE STATISTICS |
ESTIMATE STATISTICS SAMPLE estimate_percent PERCENT
```

See Also: For more information about the COMPUTE STATISTICS clause, see the *Oracle8i SQL Reference*.

Gathering New Optimizer Statistics

Before gathering new statistics for a particular schema, use the DBMS_STATS.EXPORT_SCHEMA_STATS procedure to extract and save existing statistics. Then, use DBMS_STATS.GATHER_SCHEMA_STATS to gather new statistics. You can implement both of these with a single call to the GATHER_SCHEMA_STATS procedure.

If key SQL statements experience significant performance degradation, then either gather statistics again using a larger sample size, or perform the following steps:

1. Use DBMS_STATS.EXPORT_SCHEMA_STATS to save the new statistics.
2. Use DBMS_STATS.IMPORT_SCHEMA_STATS to restore the old statistics. The application is now ready to run again.

You may want to use the new statistics if they result in improved performance for the majority of SQL statements, and if the number of problem SQL statements is small. In this case, do the following:

1. Create a stored outline for each problematic SQL statement using the old statistics.

   See Also: Stored outlines are pre-compiled execution plans that Oracle can use to mimic proven application performance characteristics. For more information, see Chapter 10, "Using Plan Stability".

2. Use DBMS_STATS.IMPORT_SCHEMA_STATS to restore the new statistics. Your application is now ready to run with the new statistics. However, you will continue to achieve the previous performance levels for the problem SQL statements.

Gathering Automated Statistics

You can automatically gather statistics or create lists of tables that have stale or no statistics.

To automatically gather statistics, run the DBMS_STATS.GATHER_SCHEMA_STATS and DBMS_STATS.GATHER_DATABASE_STATS procedures with the OPTIONS and objlist parameters. Use the following values for the options parameter:

GATHER STALE	Gathers statistics on tables with stale statistics.
GATHER	Gathers statistics on all tables. (default)
GATHER EMPTY	Gathers statistics only on tables without statistics.
LIST STALE	Creates a list of tables with stale statistics.
LIST EMPTY	Creates a list of tables that do not have statistics.

The objlist parameter identifies an output parameter for the LIST STALE and LIST EMPTY options. The objlist parameter is of type DBMS_STATS.OBJECTTAB.

Designating Tables for Monitoring and Automated Statistics Gathering To automatically gather statistics for a particular table, enable the monitoring attribute using the MONITORING keyword. This keyword is part of the CREATE TABLE and ALTER TABLE statement syntax.

After it is enabled, Oracle monitors the table for DML activity. This includes the approximate number of inserts, updates, and deletes for that table since the last time statistics were gathered. Oracle uses this data to identify tables with stale statistics.

View the data Oracle obtains from monitoring these tables by querying the USER_TAB_MODIFICATIONS view.

Note: There may be a few hours delay while Oracle propagates information to this view.

To disable monitoring of a table, use the NOMONITORING keyword.

See Also: For more information about the CREATE TABLE and ALTER TABLE syntax and the MONITORING and NOMONITORING keywords, see the *Oracle8i SQL Reference.*

Enabling Automated Statistics Gathering The GATHER STALE option only gathers statistics for tables that have stale statistics and for which you have enabled the MONITORING attribute. To enable monitoring for tables, use the MONITORING keyword of the CREATE TABLE and ALTER TABLE statements, as described in "Designating Tables for Monitoring and Automated Statistics Gathering" on page 8-9.

The GATHER STALE option maintains up-to-date statistics for the cost-based optimizer. Using this option at regular intervals also avoids the overhead associated with using the ANALYZE statement on all tables at one time. The GATHER option can incur much more overhead, because this option generally gathers statistics for a greater number of tables than GATHER STALE.

Use a script or job scheduling tool for the GATHER_SCHEMA_STATS and GATHER_DATABASE_STATS procedures to establish a frequency of statistics collection that is appropriate for your application. The frequency of collection intervals should balance the task of providing accurate statistics for the optimizer against the processing overhead incurred by the statistics collection process.

Creating Lists of Tables with Stale or No Statistics You can use the GATHER_SCHEMA_STATS and GATHER_DATABASE_STATS procedures to create a list of tables with stale statistics. Use this list to identify tables for which you want to manually gather statistics.

You can also use these procedures to create a list of tables with no statistics. Use this list to identify tables for which you want to gather statistics, either automatically or manually.

Preserving Versions of Statistics

You can preserve versions of statistics for tables by specifying the stattab, statid, and statown parameters in the DBMS_STATS package. Use stattab to identify a destination table for archiving previous versions of statistics. Further identify these versions using statid to denote the date and time the version was made. Use statown to identify a destination schema if it is different from the schema(s) of the actual tables. You must first create such a table using the CREATE_STAT_TABLE procedure of the DBMS_STATS package.

> **See Also:** For more information on DBMS_STATS procedures and parameters, see *Oracle8i Supplied PL/SQL Packages Reference.*

Statistics Data

Statistics includes the following data:

- Physical description of tables, for example, columns:
  - NUM_ROWS
  - BLOCKS
  - AVGRLEN
- Descriptions of attributes, for example, columns:
  - DISTINCT VALUES
  - LOWVAL
  - HIGHVAL

Data Distribution

These attributes help you determine how the data is distributed across your tables. The optimizer assumes that the data is uniformly distributed. The actual data distribution in your tables can be easily analyzed by viewing the appropriate

dictionary table, as described in DBA_TABLES for tables and DBA_TAB_COLUMNS for column statistics.

Attribute Skew

Histograms can be used to determine attribute skew. Descriptions of available access methods, for example, columns:

- HEIGHT
- LEAF BLOCK
- DISTINCT KEYS
- CLUSTERING FACTOR
- LEAF BLOCKS PER KEY
- DB BLOCKS PER KEY

See Also: For more information on histograms, see "Using Histograms" on page 8-17.

Missing Statistics

When statistics do not exist, the optimizer uses the following default values. Table 8–2 shows the defaults you can expect when statistics are missing.

Table 8–2 Default Table and Index Values When Statistics are Missing

Statistic	Default Value Used by Optimizer
Tables	
■ Cardinality	100 rows
■ Avg. row len	20 bytes
■ No. of blocks	100
■ Remote cardinality	2000 rows
■ Remote average row length	100 bytes

Table 8–2 Default Table and Index Values When Statistics are Missing

Statistic	Default Value Used by Optimizer
Indexes	
■ Levels	1
■ Leaf blocks	25
■ Leaf blocks/key	1
■ Data blocks/key	1
■ Distinct keys	100
■ Clustering factor	800 (8*no. of blocks)

Using Statistics

This section provides guidelines on how to use and view the statistics. This includes:

- Managing Statistics
- Verifying Table Statistics
- Verifying Index Statistics
- Verifying Column Statistics
- Using Histograms

Managing Statistics

This section describes statistics tables and lists the views that display information about statistics stored in the data dictionary.

Statistics Tables

The DBMS_STATS package enables you to store statistics in a *statistics table*. You can transfer the statistics for a column, table, index, or schema into a statistics table and subsequently restore those statistics to the data dictionary. The optimizer does not use statistics that are stored in a statistics table.

Statistics tables enable you to experiment with different sets of statistics. For example, you can back up a set of statistics before you delete them, modify them, or generate new statistics. You can then compare the performance of SQL statements optimized with different sets of statistics, and if the statistics stored in a table give the best performance, you can restore them to the data dictionary.

A statistics table can keep multiple distinct sets of statistics, or you can create multiple statistics tables to store distinct sets of statistics separately.

Viewing Statistics

You can use the DBMS_STATS package to view the statistics stored in the data dictionary or in a statistics table.

You can also query these data dictionary views for statistics in the data dictionary:

- USER_TABLES, ALL_TABLES, and DBA_TABLES
- USER_TAB_COLUMNS, ALL_TAB_COLUMNS, and DBA_TAB_COLUMNS
- USER_INDEXES, ALL_INDEXES, and DBA_INDEXES
- USER_CLUSTERS and DBA_CLUSTERS
- USER_TAB_PARTITIONS, ALL_TAB_PARTITIONS, and DBA_TAB_PARTITIONS
- USER_TAB_SUBPARTITIONS, ALL_TAB_SUBPARTITIONS, and DBA_TAB_SUBPARTITIONS
- USER_IND_PARTITIONS, ALL_IND_PARTITIONS, and DBA_IND_PARTITIONS
- USER_IND_SUBPARTITIONS, ALL_IND_SUBPARTITIONS, and DBA_IND_SUBPARTITIONS
- USER_PART_COL_STATISTICS, ALL_PART_COL_STATISTICS, and DBA_PART_COL_STATISTICS
- USER_SUBPART_COL_STATISTICS, ALL_SUBPART_COL_STATISTICS, and DBA_SUBPART_COL_STATISTICS

> **See Also:** For information on the statistics in these views, see *Oracle8i Reference*.

Verifying Table Statistics

To verify that the table statistics are available, execute the following statement against DBA_TABLES:

```
SQL> SELECT TABLE_NAME, NUM_ROWS, BLOCKS, AVG_ROW_LEN,
       TO_CHAR(LAST_ANALYZED, 'MM/DD/YYYY HH24:MI:SS')
     FROM DBA_TABLES
     WHERE TABLE_NAME IN ('SO_LINES_ALL','SO_HEADERS_ALL');
```

This returns the following typical data:

```
TABLE_NAME                NUM_ROWS   BLOCKS  AVH_ROW_LEN   LAST_ANALYZED
------------------------  --------   ------- -----------   -------------
SO_HEADERS_ALL             1632264   207014          449   07/29/1999 00:59:51
SO_LINES_ALL              10493845  1922196          663   07/29/1999 01:16:09
```

Verifying Index Statistics

To verify that index statistics are available and assist you in determining which are the best indexes to use in your application, execute the following statement against the dictionary DBA_INDEXES table:

```
SQL> SELECT INDEX_NAME "NAME", NUM_ROWS, DISTINCT_KEYS "DISTINCT",
            LEAF_BLOCKS, CLUSTERING_FACTOR "CF",
            AVG_LEAF_BLOCKS_PER_KEY "ALFBPKEY"
     FROM DBA_INDEXES
     WHERE TABLE_NAME ="AP_INVOICES_ALL"
     ORDER BY INDEX_NAME;
```

This returns the following typical data:

```
NAME                    NUM_ROWS DISTINCT LEAF_BLOCKS      CF   ALFBKEY
---------------------- ---------- -------- -----------  -------  -------
AP_INVOICES_N1              18941    80712      17060   431230        1
AP_INVOICES_N3              14995        2      21403   186450    10701
AP_INVOICES_N4              13196   439859      18669  2889596        1
AP_INVOICES_N5               9734      291      24145  1543140       82
AP_INVOICES_N6              18816  1567987      22708  2579791        1
AP_INVOICES_N9               9216        3      23271   264048     7757
AP_INVOICES_U1              10892  2861077      17074   342793        1
AP_INVOICES_U2              17176  3084212      28910  2499547        1
```

Optimizer Index Determination Criteria

The optimizer uses the following criteria when determining which index to use:

- Number of rows in the index (cardinality)
- Number of distinct keys. These define the selectivity of the index.
- Level or height of the index. This indicates how deeply the data 'probe' must search in order to find the data.
- Number of leaf blocks. This is the number of I/Os needed to find the desired rows of data.

- Clustering factor (CF). This is the colocation amount of the index block relative to data blocks. The higher the CF the more likely the optimizer is to select this index.

Usage Notes

Use the following notes to assist you in deciding whether you have chosen an appropriate index for your table, data, and query:

DISTINCT Consider index `ap_invoices_n3`, the number of distinct keys, is 2. The resulting selectivity based on index `ap_invoices_n3` is poor, and the optimizer is not likely to use this index. Using this index fetches 50% of the data in the table. In this case, a full table scan is cheaper than using index `ap_invoices_n3`.

Index Cost Tie The optimizer uses alphabetic determination: If the optimizer determines that the selectivity, cost, and cardinality of two finalist indexes is the same, then it uses the two indexes' names as the deciding factor. It chooses the index with name beginning with a lower alphabetic letter or number.

Verifying Column Statistics

To verify that column statistics are available, execute the following statement against the dictionary's `DBA_TAB_COLUMNS` view:

```
SQL> SELECT COLUMN_NAME, NUM_DISTINCT, NUM_NULLS, NUM_BUCKETS, DENSITY
    FROM DBA_TAB_COLUMNS
    WHERE TABLE_NAME ="PA_EXPENDITURE_ITEMS_ALL"
    ORDER BY COLUMN_NAME;
```

This returns the following data:

```
COLUMN_NAME                    NUM_DISTINCT  NUM_NULLS NUM_BUCKETS    DENSITY
------------------------------ ------------ ---------- ----------- ----------
BURDEN_COST                            4300      71957           1 .000232558
BURDEN_COST_RATE                        675    7376401           1 .001481481
CONVERTED_FLAG                            1   16793903           1          1
COST_BURDEN_DISTRIBUTED_FLAG              2      15796           1         .5
COST_DISTRIBUTED_FLAG                     2          0           1         .5
COST_IND_COMPILED_SET_ID                 87    6153143           1 .011494253
EXPENDITURE_ID                      1171831          0           1 8.5337E-07
TASK_ID                                8648          0           1 .000115634
TRANSFERRED_FROM_EXP_ITEM_ID        1233787   15568891           1 8.1051E-07
```

Verifying column statistics are especially important for the following schema:

- Join conditions
- When the WHERE clause includes a column(s) with a bind variable, for example:

```
column x  = :variable_y
```

In these cases, the stored column statistics can be used to get a representative Cardinality estimation for the given expression.

Consider the data returned in the above example.

NUM_DISTINCT Column Statistic

Low Column CONVERTED_FLAG: NUM_DISTINCT = 1. In this case this column has only one value. If in the WHERE clause, then there is a bind variable on column CONVERTED_FLAG = :variable_y, say. If CONVERTED_FLAG is low, as the case in this example, then this leads to poor selectivity and CONVERTED_FLAG is a poor candidate to be used as the index.

Column COST_BURDEN_DISTRIBUTED_FLAG: NUM_DISTINCT = 2. Likewise, this is low. COST_BURDEN_DISTRIBUTED_FLAG is not a good candidate for index unless there is much skew. If there is data skew of, say, 90%, then 90% of the data has one particular value and 10% of the data has another value. If the query only needs to access the 10%, then a histogram would be needed on that column in order for the optimizer to recognize the skew and utilize an index on this column.

High NUM_DISTINCT is more than one million for column EXPEDITURE_ID. If there is a bind variable on column EXPENDITURE_ID, then this leads to high selectivity (implying high density of data on this column). In other words, EXPENDITURE_ID is a good candidate to be used as the index.

NUM_NULL Column Statistic

NUM_NULLS indicates the number of null statistics.

Low For example, if a single column index has few null, such as the COST_DISTRIBUTED_FLAG column, then if this column is used as the index, the resulting data set will be large.

High If there are many nulls on a particular column, such as the CONVERTED_FLAG column, then if this column is used as the index, the resulting data set will be small. This means that COST_DISTRIBUTED_FLAG would be a more appropriate column to index.

DENSITY Column Statistic

This indicates how dense the values of that column are.

Column Statistics and Join Methods

Column statistics are useful to help determine the most efficient join method, which, in turn, is also based on the number of rows returned.

Using Histograms

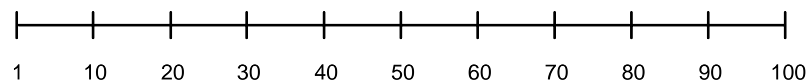
The cost-based optimizer uses data value histograms to get accurate estimates of the distribution of column data. A *histogram* partitions the values in the column into bands, so that all column values in a band fall within the same range. Histograms provide improved selectivity estimates in the presence of data skew, resulting in optimal execution plans with nonuniform data distributions.

One of the fundamental capabilities of the 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 cost-based optimizer uses height-based histograms on specified attributes to describe the distributions of nonuniform domains. In a height-based histogram, the column values are divided into bands so that each band contains approximately the same number of values. The useful information that the histogram provides, then, is where in the range of values the endpoints fall.

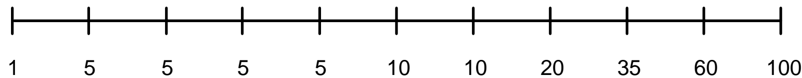
> **See Also:** For more information, see "Types of Histograms" on page 8-19.

Consider a column C with values between 1 and 100 and a histogram with 10 buckets. If the data in C is uniformly distributed, then the 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, then 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.

When to Use Histograms

Histograms can affect performance and should be used only when they substantially improve query plans. In general, you should create histograms on columns that are frequently used in `WHERE` clauses of queries and have a highly skewed data distribution. For many applications, it is appropriate to create histograms for all indexed columns because indexed columns typically are the columns most often used in `WHERE` clauses.

Histograms are persistent objects, so there is a maintenance and space cost for using them. You should compute histograms only for columns that you know have highly skewed data distribution. For uniformly distributed data, the cost-based optimizer can make fairly accurate guesses about the cost of executing a particular statement without the use of histograms.

Histograms, like all other optimizer statistics, are static. They are useful only when they reflect the current data distribution of a given column. (The data in the column can change as long as the *distribution* remains constant.) If the data distribution of a column changes frequently, you must recompute its histogram frequently.

Histograms are *not* useful for columns with the following characteristics:

- All predicates on the column use bind variables.
- The column data is uniformly distributed.
- The column is not used in `WHERE` clauses of queries.
- The column is unique and is used only with equality predicates.

Creating Histograms

You generate histograms by using the `DBMS_STATS` package or the `ANALYZE` statement. You can generate histograms for columns of a table or partition. Histogram statistics are not collected in parallel.

For example, to create a 10-bucket histogram on the SAL column of the emp table, issue the following statement:

```
EXECUTE DBMS_STATS.GATHER_TABLE_STATS
('scott','emp', METHOD_OPT => 'FOR COLUMNS SIZE 10 sal');
```

The SIZE keyword declares the maximum number of buckets for the histogram. You would create a histogram on the SAL column if there was an unusually high number of employees with the same salary and few employees with other salaries. You can also collect histograms for a single partition of a table.

> **See Also:** For more information on the DBMS_STATS package, see *Oracle8i Supplied PL/SQL Packages Reference*.

Choosing the Number of Buckets for a Histogram

The default number of buckets for a histogram is 75. This value provides an appropriate level of detail for most data distributions. However, because the number of buckets in the histogram, also known as 'the sampling rate', and the data distribution both affect a histogram's usefulness, you may need to experiment with different numbers of buckets to obtain optimal results.

If the number of frequently occurring distinct values in a column is relatively small, then set the number of buckets to be greater than that number.

Types of Histograms

There are two types of histograms:

- Height-Based Histograms
- Value-Based Histograms

Height-Based Histograms

Height-based histograms place approximately the same number of values into each range, so that the endpoints of the range are determined by how many values are in that range.

Consider that a table's query results in the following four sample values: 4, 18, 30, and 35.

For a height-based histogram, we consider each of these values to occupy a portion of one bucket, in proportion to their size. The resulting selectivity is computed with the following formula:

```
S = Height(35)/ Height(4 + 18 + 30 + 35)
```

Value-Based Histograms

Consider the same four sample values in the example above. In a value-based histogram a bucket is used to represent each of the four distinct values. In other words, one bucket represents 4, one bucket represents 18, another represents 30, and another represents 35. The resulting selectivity is computed with the following formula:

```
S = [#rows (35)/(#rows(4) + #rows(18) + #rows(30) + #rows(35))]/ #buckets
```

If there are many different values anticipated for a particular column of your table, it is preferable to use the value-based histogram rather than the height-based histogram. This is because if there is much data skew in the height, then the skew can offset the selectivity calculation and give a non-representative selectivity value.

Histogram Example

The following example illustrates the use of a histogram in order to improve the execution plan and demonstrate the skewed behavior of the `s6` indexed column.

```
UPDATE so_lines l
SET open_flag=null,
     s6=10,
     s6_date=sysdate,
WHERE l.line_type_code in ('REGULAR','DETAIL','RETURN') AND
      l.open_flag||'' = 'Y'AND NVL(l.shipped_quantity, 0)=0 OR
      NVL(l.shipped_quantity, 0) != 0 AND
     l.shipped_quantity +NVL(l.cancelled_quantity, 0)= l.ordered_quantity)) AND
      l.s6=18
```

This query shows the skewed distribution of data values for `s6`. In this case, there are two distinct non-null values: 10 and 18. The majority of the rows consists of `s6` = 10 (1,589,464), while a small amount of rows consist of `s6` = 18 (13,091).

```
S6:      COUNT(*)
======================
10        1,589,464
18           13,091
NULL         21,889
```

The selectivity of column `s6`, where `s6` = 18:

S = 13,091 / (13,091 + 1,589,464) = 0.008

- *If No Histogram is Used*: Then the selectivity of column `s6` is assumed to be 50%, uniformly distributed across 10 and 18. This is not selective; therefore, `s6` is not an ideal choice for use as an index.
- *If a Histogram is Used*: Then the data distribution information is stored in the dictionary. This allows the optimizer to use this information and compute the correct selectivity based on the data distribution. In the above example, the selectivity, based on the histogram data, is 0.008. This a relatively high, or good, selectivity, which indicates to the optimizer to use an index on column `s6` in the execution plan.

Viewing Histograms

You can view histogram information with the following data dictionary views:

- `USER_HISTOGRAMS`, `ALL_HISTOGRAMS`, and `DBA_HISTOGRAMS`
- `USER_PART_HISTOGRAMS`, `ALL_PART_HISTOGRAMS`, and `DBA_PART_HISTOGRAMS`
- `USER_SUBPART_HISTOGRAMS`, `ALL_SUBPART_HISTOGRAMS`, and `DBA_SUBPART_HISTOGRAMS`
- `TAB_COLUMNS`

Number of Rows View the following `DBA_HISTOGRAMS` dictionary table for the number of buckets; i.e., the number of rows, for each column:

- `ENDPOINT_NUMBER`
- `ENDPOINT_VALUE`

> **See Also:** For column descriptions of data dictionary views, as well as histogram use and restrictions, see *Oracle8i Reference.*

Verifying Histogram Statistics

To verify that histogram statistics are available, execute the following statement against the dictionary's `DBA_HISTOGRAMS` table:

```
SQL> SELECT ENDPOINT_NUMBER, ENDPOINT_VALUE
     FROM DBA_HISTOGRAMS
     WHERE TABLE_NAME ="SO_LINES_ALL" AND COLUMN_NAME="S2";
```

This returns the following typical data:

```
ENDPOINT_NUMBER   ENDPOINT_VALUE
---------------  ---------------
           1365                4
           1370                5
           2124                8
           2228               18
```

Consider the difference between two ENDPOINT_NUMBER values, such as 1370 - 1365 = 5. This indicates that 5 values are represented in the bucket containing the endpoint 1365.

If endpoint numbers are very different, then this implies the use of more buckets, where one row corresponds to one bucket.

9

Optimizing SQL Statements

This chapter describes how Oracle optimizes Structured Query Language (SQL) using the cost-based optimizer (CBO).

This chapter contains the following sections:

- Approaches to SQL Statement Tuning
- Tuning Goals
- Best Practices
- SQL Tuning Tips
- Using EXISTS versus IN
- Trouble Shooting
- Tuning Distributed Queries

Note: Although some Oracle tools and applications mask the use of SQL, all database operations are performed using SQL. Any other data access method circumvents the security built into Oracle and potentially compromises data security and integrity.

Approaches to SQL Statement Tuning

This section describes five ways you can improve SQL statement efficiency:

- Restructuring the Indexes
- Restructuring the Statement
- Modifying or Disabling Triggers
- Restructuring the Data
- Keeping Statistics Current and Using Plan Stability to Preserve Execution Plans

---

Note: The guidelines described in this section are oriented to production SQL that will be executed frequently. Most of the techniques that are discouraged here can legitimately be employed in ad hoc statements or in applications run infrequently where performance is not critical.

Restructuring the Indexes

Restructuring the indexes is a good starting point, because it has more impact on the application than does restructuring the statement or the data.

- Remove nonselective indexes to speed the DML.
- Index performance-critical access paths.
- Consider hash clusters, but watch uniqueness.
- Consider index clusters only if the cluster keys are similarly sized.

Do not use indexes as a panacea. Application developers sometimes think that performance will improve if they write more indexes. If a single programmer creates an appropriate index, then this might indeed improve the application's performance. However, if 50 programmers each create an index, then application performance will probably be hampered!

Restructuring the Statement

After restructuring the indexes, you can try restructuring the statement. Rewriting an inefficient SQL statement is often easier than repairing it. If you understand the purpose of a given statement, then you may be able to quickly and easily write a new statement that meets the requirement.

Consider Alternative SQL Syntax

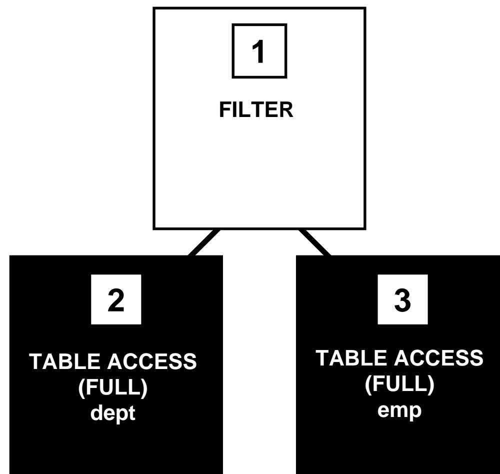
Because SQL is a flexible language, more than one SQL statement may meet the needs of your application. Although two SQL statements may produce the same result, Oracle may process one faster than the other. You can use the results of the `EXPLAIN PLAN` statement to compare the execution plans and costs of the two statements and determine which is more efficient.

This example shows the execution plans for two SQL statements that perform the same function. Both statements return all the departments in the `dept` table that have no employees in the `emp` table. Each statement searches the `emp` table with a subquery. Assume there is an index, `deptno_index`, on the `deptno` column of the `emp` table.

The first statement and its execution plan:

```
SELECT dname, deptno
   FROM dept
   WHERE deptno NOT IN
      (SELECT deptno FROM emp);
```

Figure 9–1 Execution Plan with Two Full Table Scans

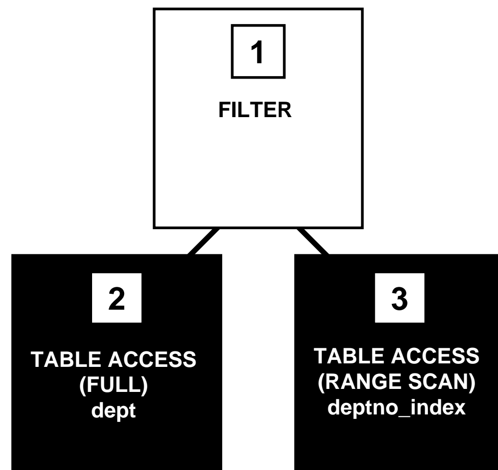


Step 3 of the output indicates that Oracle executes this statement by performing a full table scan of the `emp` table despite the index on the `deptno` column. This full table scan can be a time-consuming operation. Oracle does not use the index, because the subquery that searches the `emp` table does not have a `WHERE` clause that makes the index available.

However, this SQL statement selects the same rows by accessing the index:

```
SELECT dname, deptno
FROM dept
WHERE NOT EXISTS
    (SELECT deptno
    FROM emp
    WHERE dept.deptno = emp.deptno);
```

Figure 9–2 Execution Plan with a Full Table Scan and an Index Scan



The `WHERE` clause of the subquery refers to the `deptno` column of the `emp` table, so the index `deptno_index` is used. The use of the index is reflected in step 3 of the execution plan. The index range scan of `deptno_index` takes less time than the full scan of the `emp` table in the first statement. Furthermore, the first query performs one full scan of the `emp` table for every `deptno` in the `dept` table. For these reasons, the second SQL statement is faster than the first.

If you have statements in your applications that use the `NOT IN` operator, as the first query in this example does, then you should consider rewriting them so that they use the `NOT EXISTS` operator. This would allow such statements to use an index, if one exists.

Note: Alternative SQL syntax is effective only with the rule-based optimizer.

Compose Predicates Using AND and =

Use equijoins whenever possible. Without exception, statements that perform equijoins on untransformed column values are the easiest to tune.

Choose an Advantageous Join Order

Join order can have a significant effect on performance. The main objective of SQL tuning is to avoid performing unnecessary work to access rows that do not affect the result. This leads to three general rules:

- Avoid a full-table scan if it is more efficient to get the required rows through an index.
- Avoid using an index that fetches 10,000 rows from the driving table if you could instead use another index that fetches 100 rows.
- Choose the join order so as to join fewer rows to tables later in the join order.

The following example shows how to tune join order effectively:

```
SELECT info
FROM taba a, tabb b, tabc c
WHERE a.acol BETWEEN :alow AND :ahigh
    AND b.bcol BETWEEN :blow AND :bhigh
    AND c.ccol BETWEEN :clow AND :chigh
    AND a.key1 = b.key1
    AND a.key2 = c.key2;
```

1. Choose the driving table and the driving index (if any).

   The first three conditions in the example above are filter conditions applying to only a single table each. The last two conditions are join conditions.

   Filter conditions dominate the choice of driving table and index. In general, the driving table should be the one containing the filter condition that eliminates the highest percentage of the table. Thus, if the range of :`alow` to :`ahigh` is narrow compared with the range of `acol`, but the ranges of :`b`* and :`c`* are relatively large, then `taba` should be the driving table, all else being equal.

2. Choose the right indexes.

   After you know your driving table, choose the most selective index available to drive into that table. Alternatively, choose a full table scan if that would be more efficient. From there, the joins should all happen through the join indexes, the indexes on the primary or foreign keys used to connect that table to an earlier table in the join tree. Rarely should you use the indexes on the non-join

conditions, except for the driving table. Thus, after `taba` is chosen as the driving table, you should use the indexes on `b.key1` and `c.key2` to drive into `tabb` and `tabc`, respectively.

3. Choose the best join order, driving to the best unused filters earliest.

   The work of the following join can be reduced by first joining to the table with the best still-unused filter. Thus, if "`bcol BETWEEN` ..." is more restrictive (rejects a higher percentage of the rows seen) than "`ccol between` ...", the last join can be made easier (with fewer rows) if `tabb` is joined before `tabc`.

Use Untransformed Column Values

Use untransformed column values. For example, use:

```
WHERE a.order_no = b.order_no
```

Rather than:

```
WHERE TO_NUMBER (SUBSTR(a.order_no, instr(b.order_no, '.') - 1))
= TO_NUMBER (SUBSTR(a.order_no, instr(b.order_no, '.') - 1))
```

Do not use SQL functions in predicate clauses or `WHERE` clauses. The use of an aggregate function, especially in a subquery, often indicates that you could have held a derived value on a master record.

Avoid Mixed-Type Expressions

Avoid mixed-mode expressions, and beware of implicit type conversions. When you want to use an index on the `VARCHAR2` column `charcol`, but the `WHERE` clause looks like this:

```
AND charcol = <numexpr>
```

Where `numexpr` is an expression of number type (for example, 1, `USERENV`('`SESSIONID`'), `numcol`, `numcol`+0,...), Oracle translates that expression into:

```
AND TO_NUMBER(charcol) = numexpr
```

This has the following consequences:

- Any expression using a column, such as a function having the column as its argument, causes the optimizer to ignore the possibility of using an index on that column, even a unique index.

- If the system processes even a single row having `charcol` as a string of characters that does not translate to a number, then an error is returned.

You can avoid this problem by replacing the top expression with the explicit conversion:

```
AND charcol = TO_CHAR(<numexpr>)
```

Alternatively, make all type conversions explicit. The statement:

```
numcol = charexpr
```

allows use of an index on `numcol`, because the default conversion is always character-to-number. This behavior, however, is subject to change. Making type conversions explicit also makes it clear that `charexpr` should always translate to a number.

Write Separate SQL Statements for Specific Values

SQL is not a procedural language. Using one piece of SQL to do many different things is not a good idea: it usually results in a less-than-optimal result for each task. If you want SQL to accomplish different things, then write two different statements rather than writing one statement that will do different things depending on the parameters you give it.

Optimization (determining the execution plan) takes place before the database knows what values will be substituted into the query. An execution plan should not, therefore, depend on what those values are. For example:

```
SELECT info
FROM tables
WHERE ...
    AND somecolumn BETWEEN DECODE(:loval, 'ALL', somecolumn, :loval)
    AND DECODE(:hival, 'ALL', somecolumn, :hival);
```

Written as shown, the database cannot use an index on the `somecolumn` column, because the expression involving that column uses the same column on both sides of the `BETWEEN`.

This is not a problem if there is some other highly selective, indexable condition you can use to access the driving table. Often, however, this is not the case. Frequently, you may want to use an index on a condition like that shown, but need to know the values of `:loval`, and so on, in advance. With this information, you can rule out the `ALL` case, which should *not* use the index.

If you want to use the index whenever real values are given for `:loval` and `:hival` (that is, if you expect narrow ranges, even ranges where `:loval` often equals `:hival`), then you can rewrite the example in the following logically equivalent form:

```
SELECT /* change this half of union all if other half changes */ info
FROM tables
WHERE ...
    AND somecolumn BETWEEN :loval AND :hival
    AND (:hival != 'ALL' AND :loval != 'ALL')
UNION ALL

SELECT /* Change this half of union all if other half changes. */ info
FROM tables
WHERE ...
    AND (:hival = 'ALL' OR :loval = 'ALL');
```

If you run `EXPLAIN PLAN` on the new query, then you seem to get both a desirable and an undesirable execution plan. However, the first condition the database evaluates for either half of the `UNION ALL` is the combined condition on whether `:hival` and `:loval` are `ALL`. The database evaluates this condition before actually getting any rows from the execution plan for that part of the query.

When the condition comes back false for one part of the `UNION ALL` query, that part is not evaluated further. Only the part of the execution plan that is optimum for the values provided is actually carried out. Because the final conditions on `:hival` and `:loval` are guaranteed to be mutually exclusive, then only one half of the `UNION ALL` actually returns rows. (The `ALL` in `UNION ALL` is logically valid because of this exclusivity. It allows the plan to be carried out without an expensive sort to rule out duplicate rows for the two halves of the query.)

Use Hints to Control Access Paths

Use optimizer hints, such as `/*+ORDERED */` to control access paths. This is a better approach than using traditional techniques or "tricks of the trade" such as `CUST_NO` + 0. For example, use

```
SELECT /*+ FULL(emp) */ e.ename
FROM emp e
WHERE e.job = 'CLERK';
```

rather than

```
SELECT e.ename FROM emp e
WHERE e.job || '' = 'CLERK';
```

See Also: For more information on hints, see Chapter 7, "Using Optimizer Hints".

Use Care When Using IN and NOT IN with a Subquery

Remember that WHERE (NOT) EXISTS is a useful alternative.

Note: (NOT) EXISTS is not always equivalent to NOT IN.

Use Care When Embedding Data Value Lists in Applications

Data value lists are generally a sign that an entity is missing. For example:

```
WHERE transport IN ('BMW', 'CITROEN', 'FORD', HONDA')
```

The real objective in the WHERE clause above is to determine whether the mode of transport is an automobile, and not to identify a particular make. A reference table should be available in which transport type = 'AUTOMOBILE'.

Minimize the use of DISTINCT. DISTINCT always creates a sort; all the data must be instantiated before your results can be returned.

Reduce the Number of Calls to the Database

When appropriate, use INSERT, UPDATE, or DELETE... RETURNING to select and modify data with a single call. This technique improves performance by reducing the number of calls to the database.

See Also: For syntax information on the INSERT, UPDATE, and DELETE statements, see *Oracle8i SQL Reference*.

Use Care When Managing Views

Be careful when joining views, when performing outer joins to views, and when you consider recycling views.

Use Care When Joining Views The shared SQL area in Oracle reduces the cost of parsing queries that reference views. In addition, optimizer improvements make the processing of predicates against views very efficient. Together, these factors make possible the use of views for ad hoc queries. Despite this, joins to views are not recommended, particularly joins from one complex view to another.

The following example shows a query upon a column which is the result of a GROUP BY. The entire view is first instantiated, and then the query is run against the view data.

```
CREATE VIEW dx(deptno, dname, totsal)
AS SELECT d.deptno, d.dname, e.sum(sal)
FROM emp e, dept d
    WHERE e.deptno = d.deptno
    GROUP BY deptno, dname

SELECT *
FROM dx
WHERE deptno=10;
```

Use Care When Unnesting Subqueries Setting the UNNEST_SUBQUERY session parameter to TRUE enables subquery unnesting. Subquery unnesting unnests and merges the body of the subquery into the body of the statement that contains it, allowing the optimizer to consider them together when evaluating access paths and joins.

This parameter not cost based, and it is not set by default. UNNEST_SUBQUERY first verifies if the statement is valid. If the statement is not valid, then subquery unnesting cannot proceed. The statement must then must pass a heuristic test.

The UNNEST hint checks the subquery block for validity only. If it is valid, then subquery unnesting is enabled without Oracle checking the heuristics. If you enabled subquery unnesting with the UNNEST_SUBQUERY parameter, then the NO_UNNEST hint turns it off for specific subquery blocks.

> **See Also:** For more information on unnesting nested subqueries and the conditions that make a subquery block valid, see the *Oracle8i SQL Reference*. For more information on the UNNEST and NO_UNNEST hints, see Chapter 7, "Using Optimizer Hints".

Because subquery unnesting generates views, some views will be merged in the main query block with complex view merging. When the subquery contains an aggregate function, it is a good idea to have complex view merging enabled. This allows the inline view generated by unnesting to be merged in the main query block.

Use Care When Performing Outer Joins To Views An outer join to a multi-table view can be problematic. For example, you may start with the usual emp and dept tables with indexes on e.empno, e.deptno, and d.deptno, and create the following view:

```
CREATE VIEW empdept (empno, deptno, ename, dname)
AS SELECT e.empno, e.deptno, e.ename, d.dname
    FROM dept d, emp e
    WHERE e.deptno = d.deptno(+);
```

You may then construct the simplest possible query to do an outer join into this view on an indexed column (`e.deptno`) of a table underlying the view:

```
SELECT e.ename, d.loc
FROM dept d, empdept e
    WHERE d.deptno = e.deptno(+)
    AND d.deptno = 20;
```

The following execution plan results:

```
QUERY_PLAN
--------------------------------------------
MERGE JOIN OUTER
    TABLE ACCESS BY ROWID DEPT
     INDEX UNIQUE SCAN DEPT_U1: DEPTNO
    FILTER
     VIEW EMPDEPT
      NESTED LOOPS OUTER
       TABLE ACCESS FULL EMP
       TABLE ACCESS BY ROWID DEPT
         INDEX UNIQUE SCAN DEPT_U1: DEPTNO
```

Until both tables of the view are joined, the optimizer does not know whether the view will generate a matching row. The optimizer must therefore generate *all* the rows of the view and perform a `MERGE JOIN OUTER` with all the rows returned from the rest of the query. This approach would be extremely inefficient if all you want is a few rows from a multi-table view with at least one very large table.

Solving the problem in the preceding example is relatively easy. The second reference to `dept` is not needed, so you can do an outer join straight to `emp`. In other cases, the join need not be an outer join. You can still use the view simply by getting rid of the (+) on the join into the view.

Do Not Recycle Views Beware of writing a view for one purpose and then using it for other purposes, to which it may be ill-suited. Consider this example:

```
SELECT dname
FROM dx
WHERE deptno=10;
```

You can obtain dname and deptno directly from the dept table. It would be inefficient to obtain this information by querying the DX view (which was declared earlier in the present example). To answer the query, the view would perform a join of the dept and emp tables, even though you do not need any data from the emp table.

Modifying or Disabling Triggers

Using triggers consumes system resources. If you use too many triggers, then you may find that performance is adversely affected and you may need to modify or disable them.

Restructuring the Data

After restructuring the indexes and the statement, you can consider restructuring the data.

- Introduce derived values. Avoid GROUP BY in response-critical code.
- Implement missing entities and intersection tables.
- Reduce the network load. Migrate, replicate, partition data.

The overall purpose of any strategy for data distribution is to locate each data attribute such that its value makes the minimum number of network journeys. If the current number of journeys is excessive, then moving (migrating) the data is a natural solution.

Often, however, no single location of the data reduces the network load (or message transmission delays) to an acceptable level. In this case, consider either holding multiple copies (replicating the data) or holding different parts of the data in different places (partitioning the data).

Where distributed queries are necessary, it may be effective to code the required joins with procedures either in PL/SQL within a stored procedure, or within the user interface code.

When considering a cross-network join, you can either bring the data in from a remote node and perform the join locally, or you can perform the join remotely. The option you choose should be determined by the relative volume of data on the different nodes.

Keeping Statistics Current and Using Plan Stability to Preserve Execution Plans

After you have tuned your application's SQL statements, consider maintaining statistics with the useful procedures of the DBMS_STATS package. Also consider implementing plan stability to maintain application performance characteristics despite system changes.

> **See Also:** For more information on using statistics, see Chapter 8, "Gathering Statistics". For more information on using plan stability, see Chapter 10, "Using Plan Stability".

Tuning Goals

Structured Query Language (SQL) is used to perform all database operations, although some Oracle tools and applications simplify or mask its use. This chapter provides an overview of the issues involved in tuning database operations from the SQL point-of-view.

> **See Also:** For more information about tuning PL/SQL statements, see *PL/SQL User's Guide and Reference.*

This section introduces:

- Tuning a Serial SQL Statement
- Tuning Parallel Execution
- Tuning OLTP Applications

Always approach the tuning of database operations from the standpoint of the particular goals of your application. Are you tuning serial SQL statements or parallel operations? Do you have an online transaction processing (OLTP) application or a data warehousing application?

- Data warehousing operations process high volumes of data, and they have a high correlation with the goals of parallel operations.
- OLTP applications have a large number of concurrent users, and they correlate more with serial operations.

As a result, these two divergent types of applications have contrasting goals for tuning as described in Table 9–1.

Table 9–1 Contrasting Goals for Tuning

Tuning Situation	Goal
Serial SQL Statement	Minimize resource use by the operation.
Parallel Operations	Maximize throughput for the hardware.

Tuning a Serial SQL Statement

The goal of tuning one SQL statement in isolation is: *Minimize resource use by the operation being performed.*

You can experiment with alternative SQL syntax without actually modifying your application. To do this, use the `EXPLAIN PLAN` statement with the alternative statement that you are considering, and compare the alternative statement's execution plan and cost with that of the existing one. The cost of a SQL statement appears in the `POSITION` column of the first row generated by `EXPLAIN PLAN`. You must run the application to see which statement can actually be executed more quickly.

> **See Also:** For more information, see "Approaches to SQL Statement Tuning" on page 9-2.

Tuning Parallel Execution

The goal of tuning parallel execution is: *Maximize throughput for the given hardware.*

If you have a powerful system and a massive, high-priority SQL statement to run, then parallelize the statement so that it uses all available resources.

> **Note:** Parallel execution is only available with the Oracle8*i* Enterprise Edition.

Oracle can perform the following operations in parallel:

- Parallel query
- Parallel DML (includes `INSERT`, `UPDATE`, `DELETE`; `APPEND` hint, parallel index scans)
- Parallel DDL
- Parallel recovery
- Parallel loading

- Parallel propagation (for replication)

Look for opportunities to parallelize operations in the following situations:

- Long elapsed time

  Whenever an operation you are performing in the database takes a long time, whether it is a query or a batch job, you may be able to reduce the elapsed time by using parallel operations.

- High number of rows processed

  You can split rows so that they are not all accessed by a single process.

> **See Also:** For more information about parallel execution, see *Oracle8i Concepts* and your platform-specific Oracle documentation.
>
> For information on using the following features, see *Oracle8i Data Warehousing Guide*:
>
> - Setting the degree of parallelism and enabling adaptive multi-user
> - Tuning parallel execution parameters
> - Creating indexes in parallel
> - Partitioned index scans
> - Using bulk inserts, updates, and deletes

You can also use parallel execution to access object types within an Oracle database. For example, you can use parallel execution to access Large Binary Objects (`LOBs`).

Parallel execution benefits systems if they have *all* of the following characteristics:

- Symmetric multi-processors (SMP), clusters, or massively parallel systems.
- Sufficient I/O bandwidth.
- Under-utilized or intermittently used CPUs (for example, systems where CPU usage is typically less than 30%).
- Sufficient memory to support additional memory-intensive processes, such as sorts, hashing, and I/O buffers.

If your system lacks any of these characteristics, then parallel execution *may not* significantly improve performance. In fact, parallel execution can reduce system performance on over-utilized systems or systems with small I/O bandwidth.

When to Implement Parallel Execution

Parallel execution provides the best performance improvements in decision support systems (DSS). However, online transaction processing (OLTP) systems also benefit from parallel execution; for example, parallel index creation greatly benefits ecommerce businesses where there is little scheduled downtime.

During the day, most OLTP systems should probably not use parallel execution. During off-hours, however, parallel execution can effectively process high-volume batch operations. For example, a bank might use parallelized batch programs to perform millions of updates to apply interest to accounts.

Tuning OLTP Applications

Tuning OLTP applications mostly involves tuning serial SQL statements. You should consider two design issues: use of SQL and shared PL/SQL, and use of different transaction modes.

> **See Also:** For more information on tuning data warehouse applications, see *Oracle8i Data Warehousing Guide*.

SQL and Shared PL/SQL

To minimize parsing, use bind variables in SQL statements within OLTP applications. This way, all users can share the same SQL statements while using fewer resources for parsing.

Transaction Modes

Sophisticated users can use discrete transactions if performance is of the utmost importance, and if the users are willing to design the application accordingly.

Serializable transactions can be used if the application must be ANSI compatible. Because of the overhead inherent in serializable transactions, Oracle strongly recommends the use of read-committed transactions instead.

> **See Also:** For more information, see Chapter 17, "Transaction Modes".

Triggers

If excessive use of triggers degrades system performance, then modify the conditions under which triggers fire by executing the `CREATE TRIGGER` or `CREATE OR REPLACE TRIGGER` statements. You can also turn off triggers with the `ALTER TRIGGER` statement.

Note: Excessive use of triggers for frequent events such as logons, logoffs, and error events can degrade performance, because these events affect all users.

Best Practices

This section documents the best practices for developing and tuning SQL with the cost-based optimizer (CBO). This includes the following:

- Avoiding Rule-Based Optimizer Techniques
- Index Cost
- Optimizing SQL Statements
- Avoiding Complex Expressions
- Optimizing SQL Statements
- Handling Complex Logic in the Application

Avoiding Rule-Based Optimizer Techniques

The traditional RBO tuning techniques include:

- Disabling indexes
  - col+0 or col || ‘‘
  - Wrap function around column, such as `NVL` (col, -999) or `TO_NUMBER`

  Because the CBO is cost based, it is not necessary to force or disable a particular index. The CBO chooses the access path with the best cost.
- Working the table order in the `FROM` clause.

  The CBO chooses the most efficient join order based on cost after permuting the possible join graphs. Hence, there is no need, or benefit, to ordering the `FROM` clause under the CBO.

Index Cost

In the following example, the CBO may choose a full table scan if the index probe on `employee_num` is too costly (e.g., the estimated cardinality for employees having employee numbers beginning with 20 is high).

```
SELECT employee_num, full_name NAME, employee_id
FROM mtl_employees_current_view
WHERE (employee_num LIKE '20%') AND
      (organization_id = :1)
ORDER BY employee_num;
```

Analyzing Object Statistics

The object statistics include the following:

- Column statistics
- Data skew
- Table statistics
- Index statistics
- Partition statistics

The following example illustrates the cost model and selectivity of a query which, under the RBO, used an inefficient index. The CBO chooses a more efficient plan.

```
SELECT item.expenditure_item_id
FROM pa_tasks t,
     pa_expenditures exp,
     pa_expenditure_types etype,
     pa_expenditure_items item
WHERE
TRUNC(exp.expenditure_ending_date)<=TRUNC(NVL(TO_DATE(:b0),
exp.expenditure_ending_date))
    AND exp.expenditure_status_code||''='APPROVED'
    AND exp.expenditure_group=NVL(:b1,exp.expenditure_group)
    AND exp.expenditure_id=item.expenditure_id
    AND (NVL(item.request_id,(:b2+1))<>:b2 OR item.cost_dist_rejection_code IS
NULL )
    AND item.cost_distributed_flag='N' and t.task_id=item.task_id
    AND t.project_id=DECODE(:b4,0,T.project_id,:b4)
    AND item.expenditure_type=etype.expenditure_type
    AND etype.system_linkage_function||''=:b6
 ORDER BY item.expenditure_item_date;
COST DISTRIBUTED FLAG
C                          7
N                     80,251
Y                 16,534,822
```

Rule Plan

```
Cost= SELECT STATEMENT
COUNT(*)
Cost=   SORT ORDER BY
===================================
Cost=     NESTED LOOPS
Cost=       NESTED LOOPS
Cost=         NESTED LOOPS
Cost=           TABLE ACCESS BY INDEX ROWID PA_EXPENDITURE_ITEMS_ALL
Cost=             INDEX RANGE SCAN PA_EXPENDITURE_ITEMS_N3: COST_DISTRIBUTED_
FLAG
Cost=           TABLE ACCESS BY INDEX ROWID PA_EXPENDITURE_TYPES
Cost=             INDEX UNIQUE SCAN PA_EXPENDITURE_TYPES_U1: EXPENDITURE_TYPE
Cost=         TABLE ACCESS BY INDEX ROWID PA_EXPENDITURES_ALL
Cost=           INDEX UNIQUE SCAN PA_EXPENDITURES_U1: EXPENDITURE_ID
Cost=       TABLE ACCESS BY INDEX ROWID PA_TASKS
Cost=         INDEX UNIQUE SCAN PA_TASKS_U1: TASK_ID
```

CBO Plan (default)

```
Cost=6503 SELECT STATEMENT
Cost=6503   SORT ORDER BY
Cost=6489     NESTED LOOPS
Cost=6487       NESTED LOOPS
Cost=6478         MERGE JOIN CARTESIAN
Cost=6477           TABLE ACCESS FULL PA_EXPENDITURES_ALL
Cost=1             SORT JOIN
Cost=1               TABLE ACCESS FULL PA_EXPENDITURE_TYPES
Cost=9           TABLE ACCESS BY INDEX ROWID PA_EXPENDITURE_ITEMS_ALL
Cost=4             INDEX RANGE SCAN PA_EXPENDITURE_ITEMS_N1: EXPENDITURE_ID
Cost=2         TABLE ACCESS BY INDEX ROWID PA_TASKS
Cost=1           INDEX UNIQUE SCAN PA_TASKS_U1: TASK_ID
```

Force Rule Plan Using Hints

This illustrates that the cost of the RBO plan is significantly higher than that of the the default CBO generated plan.

```
Cost=592532 SELECT STATEMENT
Cost=592532   SORT ORDER BY
Cost=592518     NESTED LOOPS
Cost=592516       NESTED LOOPS
Cost=587506         NESTED LOOPS
Cost=504831           TABLE ACCESS BY INDEX ROWID PA_EXPENDITURE_ITEMS_ALL
Cost=32573             INDEX RANGE SCAN PA_EXPENDITURE_ITEMS_N3:
```

```
Cost=1           TABLE ACCESS BY INDEX ROWID PA_EXPENDITURE_TYPES
Cost=             INDEX UNIQUE SCAN PA_EXPENDITURE_TYPES_U1:
Cost=2         TABLE ACCESS BY INDEX ROWID PA_EXPENDITURES_ALL
Cost=1           INDEX UNIQUE SCAN PA_EXPENDITURES_U1:
Cost=2       TABLE ACCESS BY INDEX ROWID PA_TASKS
Cost=1         INDEX UNIQUE SCAN PA_TASKS_U1:
```

Rewrite SQL

In order to avoid the full table scan, the query can be rewritten in order to optimize by using a more selective filter. In this case, the expenditure group is rather selective, but the `NVL()` function prevented an index from being used.

```
SELECT item.expenditure_item_id
FROM pa_tasks t,
     pa_expenditures exp,
     pa_expenditure_types etype,
     pa_expenditure_items item
WHERE
TRUNC(exp.expenditure_ending_date)<=TRUNC(NVL(TO_DATE(:b0),
exp.expenditure_ending_date))
    AND exp.expenditure_status_code||''='APPROVED'
    AND exp.expenditure_group=:b1
    AND exp.expenditure_id=item.expenditure_id
    AND (NVL(item.request_id,(:b2+1))<>:b2 OR item.cost_dist_rejection_code IS
NULL)
    AND item.cost_distributed_flag='N' and t.task_id=item.task_id
    AND t.project_id=DECODE(:b4,0,t.project_id,:b4)
    AND item.expenditure_type=etype.expenditure_type
    AND etype.system_linkage_function||''=:b6
 ORDER BY item.expenditure_item_date
```

New CBO Plan

```
Cost=32 SELECT STATEMENT
Cost=32   SORT ORDER BY
Cost=18     NESTED LOOPS
Cost=16       NESTED LOOPS
Cost=7         MERGE JOIN CARTESIAN
Cost=1           TABLE ACCESS FULL PA_EXPENDITURE_TYPES
Cost=6           SORT JOIN
Cost=6             TABLE ACCESS BY INDEX ROWID PA_EXPENDITURES_ALL
Cost=2               INDEX RANGE SCAN PA_EXPENDITURES_N3: EXPENDITURE_GROUP
Cost=9         TABLE ACCESS BY INDEX ROWID PA_EXPENDITURE_ITEMS_ALL
Cost=4           INDEX RANGE SCAN PA_EXPENDITURE_ITEMS_N1: EXPENDITURE_ID
Cost=2       TABLE ACCESS BY INDEX ROWID PA_TASKS
```

```
Cost=1          INDEX UNIQUE SCAN PA_TASKS_U1: TASK_ID
```

Note: Although there is a full table scan on the `pa_expenditure_types` table, this is only a small lookup table.

Avoiding Complex Expressions

Avoid the following kind of complex expressions:

- `col1 = NVL (:b1,col1)`
- `NVL (col1,-999) = ….`
- `TO_DATE()`, `TO_NUMBER()`, etc.

These expressions prevent the optimizer from assigning valid cardinality or selectivity estimates, and can in turn affect the overall plan and the join method.

Add the predicate versus using `NVL()` technique.

For example:

```
SELECT employee_num, full_name NAME, employee_id
FROM mtl_employees_current_view
WHERE (employee_num = NVL (:b1,employee_num)) AND (organization_id=:1)
ORDER BY employee_num;
```

Also:

```
SELECT employee_num, full_name NAME, employee_id
FROM mtl_employees_current_view
WHERE (employee_num = :b1) AND (organization_id=:1)
ORDER BY employee_num;
```

Avoiding Balloon Tactic for Coding SQL

The balloon tactic is when a developer chooses to write a single complex SQL statement which incorporates complex application and business logic, as opposed to writing a few simple queries to achieve the same results. Developing a very large complex SQL statement has performance implications in terms of sharable memory and optimization. Coding a few simple queries in place of a single complex query is a better approach, because the individual SQL statements are easier to optimize and maintain.

Oracle Forms and Reports are powerful development tools which allow application logic to be coded using PL/SQL (triggers or program units). This helps reduce the

complexity of SQL by allowing complex logic to be handled in the Forms or Reports. In addition, you can also invoke a server side PL/SQL package which performs the few SQL statements in place of a single large complex SQL statement. Because the package is a server-side unit, there are no issues surrounding client to database round-trips and network traffic.

Handling Complex Logic in the Application

Complex logic should be handled in the application via Oracle Forms triggers, PL/SQL logic, or C-Code.

For example:

```
SELECT *
FROM ar_addresses_v
WHERE (customer_id=:1)
==================================================
AR_ADDRESSES_V:
SELECT *
FROM AR_LOOKUPS L_CAT,
 FND_TERRITORIES_VL TERR,
 FND_LANGUAGES_VL LANG,
 RA_SITE_USES SU_SHIP,
 RA_SITE_USES SU_STMT,
 RA_SITE_USES SU_DUN,
 RA_SITE_USES SU_LEGAL,
 RA_SITE_USES SU_BILL,
 RA_SITE_USES SU_MARKET,
 RA_ADDRESSES ADDR
```

The following steps were taken to improve the above query, which accessed a complex view with many outer joins:

- Rewrote the SQL statement and eliminated 6 table joins.
- Added a Forms post query trigger to populate address type fields.
- Reduced the number of rows processed.

SQL Tuning Tips

Table 9–2 lists recommended tuning tips you should implement during your SQL statement design phase:

Table 9–2 SQL Tuning Tips

SQL Tuning Tip	Notes
Do the same work faster, or do less work. Tun by selectivity.	Aim to have the least rows selected. This leads to less work and less time taken by SQL execution. It also reduces parse times.
Decompose join layers.	Analyze the joins one by one and check that their use makes sense in each circumstance. See Chapter 4, "The Optimizer" .
Examine the underlying views.	If your query accesses a view, or joins with a view, then you should examine the view thoroughly to determine if the view is optimized, or if your query even needs all the complexity from the view.
Do not be afraid of full table scans, especially for small tables.	Full table scans may make sense and be cheaper than index scans in certain situations, like with smaller tables or non-selective indexes.
Examine the execution plan in detail.	Index access and NL joins may not be optimal. For example, the query could be returning too many rows for this particular join type.
Do the math for long-running queries: ▪ For example a query may need to run in 3 minutes ▪ The query joins so_lines and so_ headers table	Verify the following: ▪ selectivity of so_headers is 5% ▪ selectivity of so_lines is 15% ▪ so_headers = 1GB, so_lines = 25GB ▪ Data working set (resultant set)=3.04GB ▪ Throughput needed = 22MB/second In other words, your expectations of needing the query to run in 3 minutes could be too high, depending on the system configuration.
Monitor disk reads and buffer gets	For instructions on how to do this, see "Disk Reads and Buffer Gets" on page 9-27.
Joins * Review the outer joins * Replace join with sub-query	For advice on how to do this, see "Choose an Advantageous Join Order" on page 9-5.
Choosing `EXISTS` or `IN`	For advice on how to decide, see "Using EXISTS versus IN" on page 9-28.
Predicate collapsing	See "Predicate Collapsing" on page 9-24.
Tune for the typical case	See "Tuning for the Typical Case" on page 9-25.

Using EXPLAIN PLAN on All Queries

It is important that you generate and review execution plans for all your SQL statements to ensure optimal performance.

> **See Also:** For more information on execution plans, see Chapter 5, "Using EXPLAIN PLAN".

Predicate Collapsing

Predicate collapsing occurs when a column predicate involves more than one bind variable. An expression of the form `[col = DECODE (:b1,'',:b3,col)]` is a an example of predicate collapsing. This implies that if the bind variable 1 is null, then the bind variable 3 should be used; otherwise, the expression will result in `[ col = col]`. This prevents the optimizer from utilizing the index on the "`col`" column due to the decode construct.

The following example demonstrates how predicate collapsing is used to collapse a name bind variable with the `delivery_id` bind variable in a single filter. As can be seen from the `EXPLAIN PLAN`, this results in a full table scan on the `wsh_deliveries` table because of the `NVL()` construct on the `delivery_id` column, as well as the `DECODE()` construct on the name column.

```
SELECT delivery_id, planned_departure_id, organization_id, status_code
FROM wsh_deliveries
WHERE delivery_id = NVL(:b1,delivery_id) AND name = DECODE(:b1,'',:b3, NAME)
ORDER BY UPPER(HRE.full_name)

PLAN:

Cost=2090 SELECT STATEMENT
Cost=2090   TABLE ACCESS FULL WSH_DELIVERIES
```

This query can be rewritten using a `UNION` to short-circuit one-side of the `UNION` based on the bind variable values. For example, if the `delivery_id` bind is supplied, only the first branch of the `UNION` is executed.

If a value for the name bind variable is supplied, then the second branch of the `UNION` is executed. In either case, both sides of the `UNION` use rather selective indexes on either the `delivery_id` column or the name column. This is much more efficient than the original query which performed a full table scan.

```
SELECT delivery_id, planned_departure_id, organization_id, status_code
FROM wsh_deliveries
WHERE delivery_id = :b1 AND (:b1 IS NOT NULL)
UNION
SELECT delivery_id, planned_departure_id, organization_id, status_code
FROM wsh_deliveries
WHERE name = :b2 AND (:b1 is null)

Cost=34 SELECT STATEMENT
Cost=34   SORT UNIQUE
Cost=     UNION-ALL
Cost=       FILTER
Cost=3         TABLE ACCESS BY INDEX ROWID WSH_DELIVERIES
Cost=2           INDEX UNIQUE SCAN WSH_DELIVERIES_U1: DELIVERY_ID
Cost=       FILTER
Cost=3         TABLE ACCESS BY INDEX ROWID WSH_DELIVERIES
Cost=2           INDEX UNIQUE SCAN WSH_DELIVERIES_U2: NAME
```

Tuning for the Typical Case

The following example illustrates how a query can be optimized for the general case. Specifically, this purchasing query determines the list of approvers which can approve a purchase order for a given organizational structure. However, in most cases, the end user provides the approver name via a name pattern, and, therefore, it is not necessary to scan all the approvers.

```
SELECT COUNT(*), COUNT(DISTINCT HR.employee_id ), HR.full_name,
  HR.employee_num, HR.employee_id
FROM hr_employees_current_v HR,
    (SELECT DISTINCT PEH.superior_id
    FROM po_employee_hierarchies PEH
    WHERE PEH.position_structure_id = :1
    AND PEH.employee_id > 0) PEHV WHERE PEHV.superior_id = HR.employee_id
    AND (:2 =  'Y' OR (:3 = 'N' AND HR.employee_id != :4))
GROUP BY full_name, employee_num, employee_id
ORDER BY full_name
```

```
call     count       cpu    elapsed       disk      query  current        ros
------- ------  -------- ---------- ---------- ---------- ---------- ---------
Parse        1      0.00       0.00          0          0   0             0
Execute      1      0.00       0.00          0          0   0             0
Fetch       42     39.34      39.51       3756       7752   3            82
------- ------  -------- ---------- ---------- ---------- ----------   ---------
total       44     39.34      39.51       3756       7752   3            82
```

```
SELECT STATEMENT   GOAL: ALL_ROWS
         SORT (GROUP BY)
         FILTER
           NESTED LOOPS
            NESTED LOOPS
             VIEW
              SORT (UNIQUE)
               INDEX GOAL: ANALYZED (RANGE SCAN) OF
'PO_EMPLOYEE_HIERARCHIES_U1' (UNIQUE)
             TABLE ACCESS GOAL: ANALYZED (BY INDEX ROWID) OF
'PER_ALL_PEOPLE_F'
              INDEX (RANGE SCAN) OF 'PER_PEOPLE_F_PK' (UNIQUE)
            TABLE ACCESS GOAL: ANALYZED (BY INDEX ROWID) OF
'PER_ALL_ASSIGNMENTS_F'
             INDEX GOAL: ANALYZED (RANGE SCAN) OF
'PER_ASSIGNMENTS_F_N12' (NON-UNIQUE)
           SORT (AGGREGATE)
            TABLE ACCESS GOAL: ANALYZED (FULL) OF
'FINANCIALS_SYSTEM_PARAMS_ALL'


SELECT COUNT(*), COUNT(DISTINCT HR.employee_id ), HR.full_name,
 HR.employee_num, HR.employee_id
FROM hr_employees_current_v HR
WHERE (full_name LIKE NVL(:1,'')||'%')
AND (NVL(:2, 'N') = 'Y' OR (NVL(:3,'N') = 'N'
AND HR.employee_id !=:4)) AND EXISTS
    (SELECT PEH.superior_id
    FROM po_employee_hierarchies PEH
    WHERE PEH.position_structure_id = :5
    AND PEH.superior_id = HR.employee_id)
GROUP BY full_name, employee_num, employee_id
ORDER BY full_name
```

```
call     count       cpu    elapsed       disk      query   current        ros
------- ------  -------- ---------- ---------- ---------- ----------   ---------
Parse        0      0.00       0.00          0          0    0           0
Execute      1      0.00       0.01          0          0    0           0
Fetch        1      0.03       0.09         29         39    3           2
------- ------  -------- ---------- ---------- ---------- ----------   ---------
total        2      0.03       0.10         29         39    3           2
```

```
SELECT STATEMENT   GOAL: ALL_ROWS
  SORT (GROUP BY)
   FILTER
    NESTED LOOPS
     TABLE ACCESS   GOAL: ANALYZED (BY INDEX ROWID) OF'PER_ALL_PEOPLE_F'

      INDEX   GOAL: ANALYZED (RANGE SCAN) OF'PER_PEOPLE_F_N54'
(NON-UNIQUE)
     TABLE ACCESS   GOAL: ANALYZED (BY INDEX ROWID)
OF'PER_ALL_ASSIGNMENTS_F'
      INDEX   GOAL: ANALYZED (RANGE SCAN) OF'PER_ASSIGNMENTS_F_N12'
(NON-UNIQUE)
    TABLE ACCESS   GOAL: ANALYZED (BY INDEX ROWID) OF
'PO_EMPLOYEE_HIERARCHIES_ALL'
     INDEX   GOAL: ANALYZED (RANGE SCAN) OF 'PO_EMPLOYEE_HIERARCHIES_N2'
(NON-UNIQUE)
    SORT (AGGREGATE)
     TABLE ACCESS   GOAL: ANALYZED (FULL)
OF'FINANCIALS_SYSTEM_PARAMS_ALL'
```

Disk Reads and Buffer Gets

Monitor disk reads and buffer gets by executing the following statement:

```
SQL>  set autotrace on [explain] [stat]
```

Typical results returned are shown as follows:

```
Statistics
----------------------------------------------------------
         70  recursive calls
          0  db block gets
        591  consistent gets
        404  physical reads
          0  redo size
        315  bytes sent via SQL*Net to client
        850  bytes received via SQL*Net from client
          3  SQL*Net roundtrips to/from client
          3  sorts (memory)
          0  sorts (disk)
          0  rows processed
```

If 'consistent gets' or 'physical reads' are high relative to the amount of data returned, then this a sign that the query is expensive and needs to be reviewed for optimization.

For example, if you are expecting less than 1,000 rows back and 'consistent gets' is 1,000,000 and 'physical reads' is 10,000, then this query needs to be further optimized.

Using EXISTS versus IN

This section describes when to use `EXISTS` and when to use the `IN` clause in sub-queries.

Using EXISTS in a SELECT Statement

```
SELECT COUNT(*)
FROM so_picking_lines_all pl
WHERE (EXISTS (SELECT pld.picking_line_id
               FROM so_picking_line_details pld
               WHERE (pld.picking_line_id=pl.picking_line_id AND
                      pld.delivery_id=:b1))
      AND nvl(PL.SHIPPED_QUANTITY,0)>0)
```

Plan:

```
Cost=97740 SELECT STATEMENT
Cost=   SORT AGGREGATE
Cost=     FILTER
Cost=97740       TABLE ACCESS FULL SO_PICKING_LINES_ALL
Cost=4       TABLE ACCESS BY INDEX ROWID SO_PICKING_LINE_DETAILS
Cost=3         INDEX RANGE SCAN SO_PICKING_LINE_DETAILS_N3:
```

In this example, the use of `EXISTS` results in a full table scan because there is no selective criteria on the outer query. In this case, an `IN` operator is more appropriate. The `IN` operator enables Oracle to drive off of the `delivery_id` index, which is rather selective.

Using IN in a SELECT Statement with Nested Loop Join

```
SELECT COUNT(*)
FROM so_picking_lines_all pl
WHERE pl.picking_line_id in (SELECT pld.picking_line_id
                             FROM so_picking_line_details pld
                             WHERE pld.delivery_id=:b1)
      AND PL.SHIPPED_QUANTITY>0
```

Plan:

```
Cost=265 SELECT STATEMENT
Cost=   SORT AGGREGATE
Cost=265     NESTED LOOPS
Cost=19       VIEW
Cost=19         SORT UNIQUE
Cost=4           TABLE ACCESS BY INDEX ROWID SO_PICKING_LINE_DETAILS
Cost=3             INDEX RANGE SCAN SO_PICKING_LINE_DETAILS_N3:
Cost=2       TABLE ACCESS BY INDEX ROWID SO_PICKING_LINES_ALL
Cost=1         INDEX UNIQUE SCAN SO_PICKING_LINES_U1:
```

This is another example where `IN` is more appropriate than `EXISTS`.

Using EXISTS in an UPDATE Statement

```
UPDATE so_sales_credits_interface sc
SET request_id=:b0
WHERE request_id IS NULL AND error_flag IS NULL AND
      interface_status IS NULL AND
      EXISTS (SELECT NULL
              FROM so_headers_interface i
              WHERE sc.original_system_reference=i.original_system_reference AND
                    sc.order_source_id=i.order_source_id AND i.request_id=:b0)
```

Plan:

```
Cost=1459 UPDATE STATEMENT
Cost=   UPDATE  SO_SALES_CREDITS_INTERFACE
Cost=     FILTER
Cost=1459       TABLE ACCESS FULL SO_SALES_CREDITS_INTERFACE
Cost=2       TABLE ACCESS BY INDEX ROWID SO_HEADERS_INTERFACE_ALL
Cost=1         INDEX UNIQUE SCAN SO_HEADERS_INTERFACE_U1:
```

In this example, the use of `EXISTS` results in a full table scan because there is no selective criteria on the outer query. In this case, an `IN` operator is more appropriate. The `IN` operator enables Oracle to drive off of the `request_id` index, which is rather selective.

Trouble Shooting

This section documents the steps and procedures involved with diagnosing a CBO execution plan for a given SQL statement:

- Generate SQL trace
- Review `EXPLAIN PLAN`

- Verify statistics
- Try hints to obtain correct plan

Tuning Distributed Queries

Oracle supports transparent distributed queries to access data from multiple databases. It also provides many other distributed features, such as transparent distributed transactions and a transparent, fully automatic two-phase commit. This section explains how the Oracle8*i* optimizer decomposes SQL statements and how this affects the performance of distributed queries. The section also provides guidelines on how to influence the optimizer and avoid performance bottlenecks.

This section contains the following sections:

- Remote and Distributed Queries
- Distributed Query Restrictions
- Transparent Gateways
- Optimizing Performance of Distributed Queries

Remote and Distributed Queries

If a SQL statement references one or more remote tables, then the optimizer first determines whether all remote tables are located at the same site. If all tables are located at the same remote site, then Oracle sends the entire query to the remote site for execution. The remote site sends the resulting rows back to the local site. This is called a *remote* SQL statement. If the tables are located at more than one site, then the optimizer decomposes the query into separate SQL statements to access each of the remote tables. This is called a *distributed* SQL statement. The site where the query is executed, called the *driving site*, is usually the local site.

This section describes:

- Remote Data Dictionary Information
- Remote SQL Statements
- Distributed SQL Statements
- EXPLAIN PLAN and SQL Decomposition
- Partition Views

Remote Data Dictionary Information

If a SQL statement references multiple tables, then the optimizer must determine which columns belong to which tables before it can decompose the SQL statement. For example:

```
SELECT dname, ename
FROM dept, emp@remote
WHERE dept.deptno = emp.deptno
```

The optimizer must first determine that the `dname` column belongs to the `dept` table and the `ename` column to the `emp` table. After the optimizer has the data dictionary information of all remote tables, it can build the decomposed SQL statements.

Column and table names in decomposed SQL statements appear between double quotes. You must enclose in double quotes any column and table names that contain special characters, reserved words, or spaces.

This mechanism also replaces an asterisk (*) in the select list with the actual column names. For example:

```
SELECT *
FROM dept@remote;
```

Results in the decomposed SQL statement

```
SELECT a1."DEPTNO", a1."DNAME", a1."LOC"
FROM "DEPT" a1;
```

Note: For simplicity, double quotes are not used in the remainder of this chapter.

Remote SQL Statements

If the entire SQL statement is sent to the remote database, then the optimizer uses table aliases `A1`, `A2`, and so on, for all tables and columns in the query, in order to avoid possible naming conflicts. For example:

```
SELECT dname, ename
FROM dept@remote, emp@remote
WHERE dept.deptno = emp.deptno;
```

This is sent to the remote database as the following:

```
SELECT a2.dname, a1.ename
FROM dept a2, emp a1
WHERE a1.deptno = a2.deptno;
```

Distributed SQL Statements

When a query accesses data on one or more databases, one site *drives* the execution of the query. This is known as the *driving site*; it is here that the data is joined, grouped, and ordered. By default, the local Oracle server is the driving site. A hint called `DRIVING_SITE` enables you to manually specify the driving site.

The decomposition of SQL statements is important, because it determines the number of records or even tables that must be sent through the network. A knowledge of how the optimizer decomposes SQL statements can help you achieve optimum performance for distributed queries.

If a SQL statement references one or more remote tables, then the optimizer must decompose the SQL statement into separate queries to be executed on the different databases. For example:

```
SELECT dname, ename
FROM dept, emp@remote
WHERE dept.deptno = emp.deptno;
```

This could be decomposed into the following:

```
SELECT deptno, dname
FROM dept;
```

Which is executed locally, and:

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

Which is sent to the remote database. The data from both tables is joined locally. All this is done automatically and transparently for the user or application.

In some cases, however, it might be better to send the local table to the remote database and join the two tables on the remote database. This can be achieved either by creating a view or by using the `DRIVING_SITE` hint. If you decide to create a view on the remote database, then a database link from the remote database to the local database is also needed.

For example (on the remote database):

```
CREATE VIEW dept_emp AS
    SELECT dname, ename
    FROM dept@local, emp
    WHERE dept.deptno = emp.deptno;
```

Next, select from the remote view instead of the local and remote tables:

```
SELECT *
FROM dept_emp@remote;
```

Now, the local `dept` table is sent through the network to the remote database, joined on the remote database with the `emp` table, and the result is sent back to the local database.

> **See Also:** For details about the `DRIVING_SITE` hint, see Chapter 7, "Using Optimizer Hints".

Rule-Based Optimization The rule-based optimizer does not have information about indexes for remote tables. It never, therefore, generates a nested loops join between a local table and a remote table with the local table as the outer table in the join. It uses either a nested loops join with the remote table as the outer table or a sort merge join, depending on the indexes available for the local table.

Cost-Based Optimization The cost-based optimizer can consider more execution plans than the rule-based optimizer. The cost-based optimizer knows whether indexes on remote tables are available, and in which cases it makes sense to use them. The cost-based optimizer considers index access of the remote tables as well as full table scans, whereas the rule-based optimizer considers only full table scans.

The particular execution plan and table access that the cost-based optimizer chooses depends on the table and index statistics. For example:

```
SELECT dname, ename
FROM dept, emp@remote
WHERE dept.deptno = emp.deptno
```

Here, the optimizer might choose the local `dept` table as the driving table, and access the remote `emp` table using an index; so the decomposed SQL statement becomes the following:

```
SELECT ename FROM emp
WHERE deptno = :1
```

This decomposed SQL statement is used for a nested loops operation.

Using Views If tables are on more than one remote site, then it can be more effective to create a view than to use the `DRIVING_SITE` hint. If not all tables are on the same remote database, then the optimizer accesses each remote table separately. For example:

```
SELECT d.dname, e1.ename, e2.job
FROM dept d, emp@remote e1, emp@remote e2
WHERE d.deptno = e1.deptno
    AND e1.mgr = e2.empno;
```

This results in the decomposed SQL statements:

```
SELECT empno, ename
FROM emp;
```

and:

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

To join the two `emp` tables remotely, create a view with the join of the remote tables on the remote database. For example (on the remote database):

```
CREATE VIEW emps AS
    SELECT e1.deptno, e1.ename, e2.job
    FROM emp e1, emp e2
    WHERE e1.mgr = e2.empno;
```

Now, select from the remote view, instead of the remote tables:

```
SELECT d.dname, e.ename, e.job
FROM dept d, emps@remote e
WHERE d.deptno = e.deptno;
```

This results in the decomposed SQL statement:

```
SELECT deptno, ename, job
FROM emps;
```

Using Hints In a distributed query, all hints are supported for local tables. For remote tables, however, you can use only join order and join operation hints. (Hints for access methods, parallel hints, and so on, have no effect.) For remote mapped queries, all hints are supported.

> **See Also:** For more information on hints for join orders and hints for join operations, see Chapter 7, "Using Optimizer Hints".

EXPLAIN PLAN and SQL Decomposition

EXPLAIN PLAN gives information not only about the overall execution plan of SQL statements, but also about the way in which the optimizer decomposes SQL statements. EXPLAIN PLAN stores information in the PLAN_TABLE table. If remote tables are used in a SQL statement, then the OPERATION column contains the value REMOTE to indicate that a remote table is referenced, and the OTHER column contains the decomposed SQL statement that will be sent to the remote database. For example:

```
EXPLAIN PLAN FOR SELECT DNAME FROM DEPT@REMOTE
SELECT OPERATION, OTHER FROM PLAN_TABLE

OPERATION OTHER
--------- -------------------------------------
REMOTE    SELECT A1."DNAME" FROM "DEPT" A1
```

Note the table alias and the double quotes around the column and table names.

> **See Also:** For more information on EXPLAIN PLAN, see Chapter 5, "Using EXPLAIN PLAN".

Partition Views

Partition views coalesce tables that have the same structure, but that contain different partitions of data. Partition views are supported for distributed databases where each partition resides on a database, and the data in each partition has common geographical properties.

When a query is executed on a partition view, and when the query contains a predicate that contains the result set to a subset of the view's partitions, the optimizer chooses a plan which skips partitions that are not needed for the query. This partition elimination takes place at run time, when the execution plan references all partitions.

Partition views were the only form of partitioning available in Oracle7 Release 7.3. They are not recommended for new applications in Oracle8*i*. Partition views that were created for Oracle7 databases can be converted to partitioned tables by using the EXCHANGE PARTITION option of the ALTER TABLE statement.

> **Note:** Oracle8*i* supports partition views only for distributed queries and for backwards compatibility with Oracle7 Release 7.3. Future releases of Oracle will not support partition views.

See Also:

- For instructions on converting partition views to partitioned tables, see *Oracle8i Administrator's Guide*.
- For instructions on migrating from partition views to partitioned tables, see *Oracle8i Migration*.
- For general information on partition views and partitioned tables, see *Oracle8i Concepts*.

Using UNION ALL to Skip Partitions There are circumstances under which a `UNION ALL` view enables the optimizer to skip partitions. The Oracle server that contains the partition view must conform to the following rules:

- The `PARTITION_VIEW_ENABLED` initialization parameter is set to `true`.
- The cost-based optimizer is used.

Note: To use the cost-based optimizer, you must analyze all tables used in the `UNION ALL` views. Alternatively, you can use a hint or set the parameter `OPTIMIZER_MODE` to `ALL_ROWS` or `FIRST_ROW`. To set `OPTIMIZER_MODE` or `PARTITION_VIEW_ENABLED`, you can also use the `ALTER SESSION` statement.

Within a `UNION ALL` view, there are multiple select statements, and each of these is called a *branch*. A `UNION ALL` view is a partition view if each select statement it defines conforms to the following rules:

- The branch has exactly one table in the `FROM` clause.
- The branch contains a `WHERE` clause that defines the subset of data from the partition that is contained in the view.
- None of the following are used within the branch: `WHERE` clause with subquery, `GROUP BY`, aggregate functions, `DISTINCT`, `ROWNUM`, or `CONNECT BY/START WITH`.
- The `SELECT` list of each branch is * or an explicit expansion of "*". The `FROM` clause should be either the base table or a view of the base table that contains all the columns in the base table.

- The column names and column datatypes for all branches in the UNION ALL view are exactly the same.
- All tables used in the branch must have indexes (if any) on the same columns and number of columns.

Partition elimination is based on column transitivity with constant predicates. The WHERE clause used in the query that accesses the partition view is pushed down to the WHERE clause of each of the branches in the UNION ALL view definition. For example:

```
SELECT * FROM emp_view
WHERE deptno=30;
```

Where the view emp_view is defined as the following:

```
SELECT * FROM emp@d10 WHERE deptno=10
    UNION ALL
SELECT * FROM emp@d20 WHERE deptno=20
    UNION ALL
SELECT * FROM emp@d30 WHERE deptno=30
    UNION ALL
SELECT * FROM emp@d40 WHERE deptno=40
```

The "WHERE deptno=30" predicate used in the query is pushed down to the queries in the UNION ALL view. For a WHERE clause such as "WHERE deptno=10 and deptno=30", the optimizer applies transitivity rules to generate an extra predicate of "10=30". This extra predicate is always false; thus, the table (emp@d10) need not be accessed.

Transitivity applies to predicates which conform to the following rules:

- The predicates in the WHERE clause for each branch are of the form:

  ```
  RELATION AND RELATION ...
  ```

  where relation is of the form

  ```
  COLUMN_NAME RELOP CONSTANT_EXPRESSION
  ```

  and relop is one of =, !=, >, >=, <, <=

  > **Note:** BETWEEN ... AND is allowed by these rules, but IN is not.

- At least one predicate in the query referencing the view exists in the same form.

EXPLAIN PLAN Output To confirm that the system recognizes a partition view, check the EXPLAIN PLAN output. The following operations appear in the OPERATIONS column of the EXPLAIN PLAN output, if a query was executed on a partition view:

VIEW	This should include the optimizer cost in the COST column.
UNION-ALL	This should specify PARTITION in the OPTION column.
FILTER	When an operation is a child of the UNION-ALL operation, this indicates that a constant predicate was generated that will always be false. The partition is eliminated.

If PARTITION does not appear in the option column of the UNION-ALL operation, then the partition view was not recognized, and no partitions were eliminated. Make sure that the UNION ALL view adheres to the rules defined in "Using UNION ALL to Skip Partitions" on page 9-36.

Partition View Example The following example shows the partition view customer partitioned into two partitions: the east database contains the East Coast customers, and the west database contains the West Coast customers.

The west database contains the following table customer_west:

```
CREATE TABLE customer_west
   ( cust_no   NUMBER CONSTRAINT CUSTOMER_WEST_PK PRIMARY KEY,
     cname     VARCHAR2(10),
     location  VARCHAR2(10)
    );
```

The east database contains the database customer_east:

```
CREATE TABLE customer_east
   ( cust_no   NUMBER CONSTRAINT CUSTOMER_EAST_PK PRIMARY KEY,
     cname     VARCHAR2(10),
     location  VARCHAR2(10)
    );
```

The following partition view is created at the east database (you could create a similar view at the west database):

```
CREATE VIEW customer AS
    SELECT *
    FROM customer_east
    WHERE location='EAST'
    UNION ALL
    SELECT *
    FROM customer_west@west
    WHERE location='WEST';
```

If you execute the following statement, then notice that the `customer_west` table in the `west` database is not accessed:

```
EXPLAIN PLAN FOR SELECT * FROM customer WHERE location='EAST';
```

Note: The `east` database still needs column name and column datatype information for the `customer_west` table; therefore, it still needs a connection to the `WEST` database. In addition, the cost-based optimizer must be used. You could do this by issuing the following statement:

```
ALTER SESSION SET OPTIMIZER_MODE=ALL_ROWS
```

As shown in the `EXPLAIN PLAN` output, the optimizer recognizes that the `customer_west` partition need not be accessed:

```
SELECT LPAD(' ',LEVEL*3-3)||OPERATION OPERATION,COST,OPTIONS,
OBJECT_NODE, OTHER
FROM PLAN_TABLE
CONNECT BY PARENT_ID = PRIOR ID
START WITH PARENT_ID IS NULL

OPERATION                 COST OPTIONS    OBJECT_NOD OTHER
------------------------- ---- ---------- ---------- -------------------------
SELECT STATEMENT             1
   VIEW                      1
      UNION-ALL                 PARTITION
         TABLE ACCESS        1 FULL
         FILTER
            REMOTE           1            WEST.WORLD SELECT "CUST_NO","CNAME",
                                                     "LOCATION" FROM "CUSTOMER
                                                     _WEST" "CUSTOMER_WEST" WH
                                                     ERE "LOCATION"='EAST' AND
                                                      "LOCATION"='WEST'
```

Distributed Query Restrictions

Distributed queries within the same version of Oracle have the following restrictions:

- The cost-based optimizer should be used for distributed queries. The rule-based optimizer does not generate nested loop joins between remote and local tables when the tables are joined with equijoins.
- In the cost-based optimizer, no more than 20 indexes per remote table are considered when generating query plans. The order of the indexes varies; if the 20-index limitation is exceeded, then random variation in query plans may result.
- Reverse indexes on remote tables are not visible to the optimizer. This can prevent nested-loop joins from being used for remote tables if there is an equijoin using a column with only a reverse index.
- The cost-based optimizer cannot recognize that a remote object is partitioned. Thus, the optimizer may generate less than optimal plans for remote partitioned objects, particularly when partition pruning would have been possible, had the object been local.
- Remote views are not merged, and the optimizer has no statistics for them. It is best to replicate all mergeable views at all sites to obtain good query plans. (See the next restriction.)
- Neither the cost-based nor the rule-based optimizer can execute joins remotely. All joins are executed at the driving site. This can affect performance for `CREATE TABLE` ... `AS SELECT` if all the tables in the select list are remote. In this case, you should create a view for the `SELECT` statement at the remote site.

Transparent Gateways

The Transparent Gateways transparently access data from a non-Oracle system (relational databases, hierarchical databases, file systems, and so on), just as if it were another Oracle database.

Optimizing Heterogeneous Distributed SQL Statements

When a SQL statement accesses data from non-Oracle systems, it is said to be a heterogeneous distributed SQL statement. To optimize heterogeneous distributed SQL statements, follow the same guidelines as for optimizing distributed SQL statements that access Oracle databases only. However, you must consider that the

non-Oracle system usually does not support all the functions and operators that Oracle8*i* supports.

The Transparent Gateways tell Oracle (at connect time) which functions and operators they do support. If the other data source does not support a function or operator, then Oracle performs that function or operator. In this case, Oracle obtains the data from the other data source and applies the function or operator locally. This affects the way in which the SQL statements are decomposed and can affect performance, especially if Oracle is not on the same machine as the other data source.

Gateways and Partition Views

You can use partition views with Oracle Transparent Gateways release 8 or higher. Make sure you adhere to the rules that are defined in "Using UNION ALL to Skip Partitions" on page 9-36. In particular:

- The cost-based optimizer must be used, by using hints or setting the parameter `OPTIMIZER_MODE` to `ALL_ROWS` or `FIRST_ROWS`.
- Indexes used for each partition must be the same. Consult your gateway-specific documentation to find out whether the gateway sends index information of the non-Oracle system to the Oracle Server. If the gateway sends index information to the optimizer, then make sure that each partition uses the same number of indexes, and that you have indexed the same columns. If the gateway does not send index information, then the Oracle optimizer is not aware of the indexes on partitions. Indexes are, therefore, considered to be the same for each partition in the non-Oracle system. If one partition resides on an Oracle server, then you cannot have an index defined on that partition.
- The column names and column datatypes for all branches in the `UNION ALL` view must be the same. Non-Oracle system datatypes are mapped onto Oracle datatypes. Make sure that the datatypes of each partition that reside in the different non-Oracle systems all map to the same Oracle datatype. To see how datatypes are mapped onto Oracle datatypes, execute a `DESCRIBE` statement in SQL*Plus.

Optimizing Performance of Distributed Queries

You can improve performance of distributed queries in several ways:

- Choose the best SQL statement.

  In many cases, there are several SQL statements which can achieve the same result. If all tables are on the same database, then the difference in performance

between these SQL statements might be minimal; but, if the tables are located on different databases, then the difference in performance might be more significant.

- Use the cost-based optimizer.

  The cost-based optimizer uses indexes on remote tables, considers more execution plans than the rule-based optimizer, and generally gives better results. With the cost-based optimizer, performance of distributed queries is generally satisfactory. Only in rare occasions is it necessary to change SQL statements, create views, or use procedural code.

- Use views.

  In some situations, views can be used to improve performance of distributed queries. For example:

  – Joining several remote tables on the remote database.

  – Sending a different table through the network.

  – Using procedural code.

  In some rare occasions, it can be more efficient to replace a distributed query by procedural code, such as a PL/SQL procedure or a precompiler program. This option is mentioned here only for completeness, not because it is often needed.

10

Using Plan Stability

This chapter describes how to use plan stability to preserve performance characteristics.

This chapter contains the following sections:

- Using Plan Stability to Preserve Execution Plans
- Plan Stability Procedures for the Cost-Based Optimizer

Using Plan Stability to Preserve Execution Plans

Plan stability prevents certain database environment changes from affecting the performance characteristics of your applications. Such changes include changes in optimizer statistics, changes to the optimizer mode settings, and changes to parameters affecting the sizes of memory structures, such as SORT_AREA_SIZE, and BITMAP_MERGE_AREA_SIZE. Plan stability is most useful when you cannot risk any performance changes in your applications.

Plan stability preserves execution plans in *stored outlines*. Oracle can create a stored outline for one or all SQL statements. The optimizer then generates equivalent execution plans from the outlines when you enable the use of stored outlines.

The plans Oracle maintains in stored outlines remain consistent despite changes to your system's configuration or statistics. Using stored outlines also stabilizes the generated execution plan if the optimizer changes in subsequent Oracle releases. You can also group outlines into categories and control which category of outlines Oracle uses to simplify outline administration and deployment.

Plan stability also facilitates migration from the rule-based optimizer to the cost-based optimizer when you upgrade to a new Oracle release.

Note: If you develop applications for mass distribution, then you can use stored outlines to ensure that all your customers access the same execution plans.

Hints and Exact Text Matching

The degree to which plan stability controls execution plans is dictated by how much Oracle's hint mechanism controls execution plans, because Oracle uses hints to record stored plans. Plan stability also relies on "exact text matching" of queries when determining whether a query has a stored outline.

There is a one-to-one correspondence between SQL text and its stored outline. If you specify a different literal in a predicate, then a different outline applies. To avoid this, replace literals in your applications with bind variables. This gives your SQL statements the exact textual match for outline sharing.

See Also: For more information on how Oracle matches SQL statements to outlines, see "Matching SQL Statements with Outlines" on page 10-3.

Plan stability relies on preserving execution plans at a point in time when performance is satisfactory. In many environments, however, attributes for datatypes such as "dates" or "order numbers" can change rapidly. In these cases, permanent use of an execution plan may result in performance degradation over time as the data characteristics change.

This implies that techniques that rely on preserving plans in dynamic environments are somewhat contrary to the purpose of using cost-based optimization. Cost-based optimization attempts to produce execution plans based on statistics that accurately reflect the state of the data. Thus, you must balance the need to control plan stability with the benefit obtained from the optimizer's ability to adjust to changes in data characteristics.

How Outlines Use Hints

An outline consists primarily of a set of hints that is equivalent to the optimizer's results for the execution plan generation of a particular SQL statement. When Oracle creates an outline, plan stability examines the optimization results using the same data used to generate the execution plan. That is, Oracle uses the input to the execution plan to generate an outline and not the execution plan itself.

> **Note:** You cannot modify an outline. The `OL$` and `OL$HINTS` tables are system tables in the sense that direct manipulation is prohibited.You can embed hints in SQL statements, but this has no effect on how Oracle uses outlines. Oracle considers a SQL statement that you revised with hints to be different from the original SQL statement stored in the outline.

Matching SQL Statements with Outlines

Oracle uses one of two scenarios when compiling SQL statements and matching them with outlines. The first scenario is that if you disable outline use by setting the system/session parameter `USE_STORED_OUTLINES` to `FALSE`, then Oracle does not attempt to match SQL text to outlines. The second scenario involves the following two matching steps.

First, if you specify that Oracle must use a particular outline category, then only outlines in that category are candidates for matching. Second, if the SQL text of the incoming statement exactly matches the SQL text in an outline in that category, then Oracle considers both texts identical, and Oracle uses the outline. Oracle considers any differences a mismatch.

Differences include spacing changes, carriage return variations, embedded hints, or even differences in comment text. These rules are identical to the rules for cursor matching.

Storing Outlines

Oracle stores outline data in the OL$ table and hint data in the OL$HINTS table. Unless you remove them, Oracle retains outlines indefinitely.

The only effect outlines have on caching execution plans is that the outline's category name is used in addition to the SQL text to identify whether the plan is in cache. This ensures that Oracle does not use an execution plan compiled under one category to execute a SQL statement that Oracle should compile under a different category.

Enabling Plan Stability

Settings for several parameters, especially those ending with the suffix "_ENABLED", must be consistent across execution environments for outlines to function properly. These parameters are:

- QUERY_REWRITE_ENABLED
- STAR_TRANSFORMATION_ENABLED
- OPTIMIZER_FEATURES_ENABLE

Creating Outlines

Oracle can automatically create outlines for all SQL statements, or you can create them for specific SQL statements. In either case, the outlines derive their input from the optimizer.

Oracle creates stored outlines automatically when you set the parameter CREATE_STORED_OUTLINES to TRUE. When activated, Oracle creates outlines for all compiled SQL statements.

Note: You must ensure that schemas in which outlines are to be created have the CREATE ANY OUTLINE privilege. Otherwise, despite having turned on the CREATE_STORED_OUTLINE parameter, you will not find outlines in your database after you run your application.

You can create stored outlines for specific statements using the CREATE OUTLINE statement.

> **See Also:** For more information on the CREATE OUTLINE statement, see the *Oracle8i SQL Reference*. For information on moving from the rule-based optimizer to the cost-based optimizer, see "Using Outlines to Move to the Cost-Based Optimizer" on page 10-8.

Using Category Names For Stored Outlines

Outlines can be categorized to simplify the management task. The CREATE OUTLINE statement allows for specification of a category, while the DEFAULT category is chosen if unspecified. Likewise, the CREATE_STORED_OUTLINES parameter lets you specify a category name, where specifying TRUE produces outlines in the DEFAULT category.

If you specify a category name using the CREATE_STORED_OUTLINES parameter, then Oracle assigns all subsequently created outlines to that category until you reset the category name. Set the parameter to FALSE to suspend outline generation.

If you set CREATE_STORED_OUTLINES to TRUE, or if you use the CREATE OUTLINE statement without a category name, then Oracle assigns outlines to the category name of DEFAULT.

> **Note:** The CREATE_STORED_OUTLINES and USE_STORED_OUTLINES parameters are system- or session-specific. They are not initialization parameters. For more information on these parameters, see the *Oracle8i SQL Reference*.

Using Stored Outlines

To use stored outlines when Oracle compiles a SQL statement, set the system parameter USE_STORED_OUTLINES to true or to a category name. If you set USE_STORED_OUTLINES to true, then Oracle uses outlines in the DEFAULT category. If you specify a category with the USE_STORED_OUTLINES parameter, then Oracle uses outlines in that category until you re-set the parameter to another category

name or until you suspend outline use by setting USE_STORED_OUTLINES to FALSE. If you specify a category name and Oracle does not find an outline in that category that matches the SQL statement, then Oracle searches for an outline in the DEFAULT category.

The designated outlines only control the compilation of SQL statements that have outlines. If you set USE_STORED_OUTLINES to false, then Oracle does not use outlines. When you set USE_STORED_OUTLINES to false and you set CREATE_STORED_OUTLINES to true, Oracle creates outlines but does not use them.

When you activate the use of stored outlines, Oracle always uses the cost-based optimizer. This is because outlines rely on hints, and to be effective, most hints require the cost-based optimizer.

Test if an outline is being used with the V$SQL view. Query the OUTLINE_CATEGORY column in conjunction with the SQL statement. If an outline was applied, then this column contains the category to which the outline belongs. Otherwise, it is NULL. For example:

```
SELECT OUTLINE_CATEGORY
FROM V$SQL
WHERE SQL_TEXT LIKE 'SELECT count(*) FROM emp%';
```

Viewing Outline Data

You can access information about outlines and related hint data that Oracle stores in the data dictionary from the following views:

- USER_OUTLINES
- USER_OUTLINE_HINTS
- ALL_OUTLINES
- ALL_OUTLINE_HINTS
- DBA_OUTLINES
- DBA_OUTLINE_HINTS

Use the following syntax to obtain outline information from the USER_OUTLINES view, where the outline category is mycat:

```
SELECT NAME, SQL_TEXT
FROM USER_OUTLINES
WHERE CATEGORY='mycat';
```

Oracle responds by displaying the names and text of all outlines in category mycat.

To see all generated hints for the outline `name1`, use the following syntax:

```
SELECT HINT
FROM USER_OUTLINE_HINTS
WHERE NAME='name1';
```

> **See Also:** If necessary, you can use the procedure to move outline tables from one tablespace to another as described in "Moving Outline Tables" on page 10-7.

Using the OUTLN_PKG Package to Manage Stored Outlines

The `OUTLN_PKG` package provides procedures used for managing stored outlines and their outline categories.

> **See Also:** For detailed information on using `OUTLN_PKG` procedures, see *Oracle8i Supplied PL/SQL Packages Reference.*

Moving Outline Tables

Oracle creates the `USER_OUTLINES` and `USER_OUTLINE_HINTS` views based on data in the `OL$` and `OL$HINTS` tables respectively. Oracle creates these tables in the `SYS` tablespace using a schema called `OUTLN`. If the outlines use too much space in the `SYS` tablespace, then you can move them. To do this, create a separate tablespace and move the outline tables into it using the following process.

1. Export the `OL$` and `OL$HINTS` tables:

   ```
   EXP OUTLN/OUTLN FILE = exp_file TABLES = 'OL$' 'OL$HINTS' SILENT=y
   ```

2. Remove the previous `OL$` and `OL$HINTS` tables:

   ```
   CONNECT OUTLN/outln_password;
   DROP TABLE OL$;
   CONNECT OUTLN/outln_password;
   DROP TABLE OL$HINTS;
   ```

3. Create a new tablespace for the tables:

   ```
   CREATE TABLESPACE outln_ts
   DATAFILE 'tspace.dat' SIZE 2MB
   DEFAULT STORAGE (INITIAL 10KB NEXT 20KB
   MINEXTENTS 1 MAXEXTENTS 999 PCTINCREASE 10) ONLINE;
   ```

4. Enter the following statement:

```
ALTER USER OUTLN DEFALUT TABLESPACE outln_ts;
```

5. Import the OL$ and OL$HINTS tables:

```
IMPORT OUTLN/outln_password
FILE=exp_file TABLES = 'OL$' 'OL$HINTS' IGNORE=y SILENT=y
```

The IMPORT statement re-creates the OL$ and OL$HINTS tables in the schema named OUTLN, but the schema now resides in a new tablespace called OUTLN_TS.

Plan Stability Procedures for the Cost-Based Optimizer

This section describes procedures you can use to significantly improve performance by taking advantage of cost-based optimizer functionality. Plan stability provides a way to preserve your system's targeted execution plans with satisfactory performance while also taking advantage of new cost-based optimizer features for the rest of your SQL statements.

Topics covered in this section are:

- Using Outlines to Move to the Cost-Based Optimizer
- RDBMS Upgrades and the Cost-Based Optimizer

Using Outlines to Move to the Cost-Based Optimizer

If your application was developed using the rule-based optimizer, then a considerable amount of effort may have gone into manually tuning the SQL statements to optimize performance. You can use plan stability to leverage the effort that has already gone into performance tuning by preserving the behavior of the application when upgrading from rule-based to cost-based optimization.

By creating outlines for an application before switching to cost-based optimization, the plans generated by the rule-based optimizer can be used, while statements generated by newly written applications developed after the switch use cost-based plans. To create and use outlines for an application, use the following process.

Note: *Carefully read this procedure and consider its implications before executing it!*

---

1. Ensure that schemas in which outlines are to be created have the CREATE ANY OUTLINE privilege. For example, from SYS:

   ```
   GRANT CREATE ANY OUTLINE TO <user-name>
   ```

2. Execute syntax similar to the following to designate, for example, the RBOCAT outline category.

   ```
   ALTER SESSION SET CREATE_STORED_OUTLINES = rbocat;
   ```

3. Run your application long enough to capture stored outlines for all important SQL statements.
4. Suspend outline generation:

   ```
   ALTER SESSION SET CREATE_STORED_OUTLINES = FALSE;
   ```

5. Gather statistics with the DBMS_STATS package.
6. Alter the parameter OPTIMIZER_MODE to CHOOSE.
7. Enter this syntax to make Oracle use the outlines in category RBOCAT:

   ```
   ALTER SESSION SET USE_STORED_OUTLINES = rbocat;
   ```

8. Run the application.

   Subject to the limitations of plan stability, access paths for this application's SQL statements should be unchanged.

> **Note:** If a query was not executed in step 2, then you can capture the old behavior of the query even after switching to cost-based optimization. To do this, change the optimizer mode to RULE, create an outline for the query, and then change the optimizer mode back to CHOOSE.

RDBMS Upgrades and the Cost-Based Optimizer

When upgrading to a new Oracle release under cost-based optimization, there is always a possibility that some SQL statements will have their execution plans changed due to changes in the optimizer. While such changes benefit performance in the vast majority of cases, you might have some applications that perform well and where you would consider any changes in their behavior to be an unnecessary risk. For such applications, you can create outlines before the upgrade using the following procedure.

Note: *Carefully read this procedure and consider its implications before executing it!*

---

1. Enter the following syntax to enable outline creation:

```
ALTER SESSION SET CREATE_STORED_OUTLINES = ALL_QUERIES;
```

2. Run the application long enough to capture stored outlines for all critical SQL statements.
3. Enter this syntax to suspend outline generation:

```
ALTER SESSION SET CREATE_STORED_OUTLINES = FALSE;
```

4. Upgrade the production system to the new version of the RDBMS.
5. Run the application.

After the upgrade, you can enable the use of stored outlines, or alternatively, you can use the outlines that were stored as a backup if you find that some statements exhibit performance degradation after the upgrade.

With the latter approach, you can selectively use the stored outlines for such problematic statements as follows:

1. For each problematic SQL statement, change the `CATEGORY` of the associated stored outline to a category name similar to this:

```
ALTER OUTLINE outline_name CHANGE CATEGORY TO problemcat;
```

2. Enter this syntax to make Oracle use outlines from the category "`problemcat`".

```
ALTER SESSION SET USE_STORED_OUTLINES = problemcat;
```

Upgrading with a Test System

A test system, separate from the production system, can be useful for conducting experiments with optimizer behavior in conjunction with an upgrade. You can migrate statistics from the production system to the test system using import/export. This may alleviate the need to fill the tables in the test system with data.

You can move outlines between the systems by category. For example, after you create outlines in the `problemcat` category, export them by category using the query-based export option. This is a convenient and efficient way to export only

selected outlines from one database to another without exporting all outlines in the source database. To do this, issue these statements:

```
EXP OUTLN/outln_password FILE=<exp-file> TABLES= 'OL$' 'OL$HINTS'
QUERY='WHERE CATEGORY="problemcat"'
```

Part III

Application Design Tools for Designers and DBAs

Part III discusses how to tune your database and the various methods you use to access data for optimal database performance. The chapters in Part 3 are:

- Chapter 11, "Overview of Diagnostic Tools"
- Chapter 12, "Data Access Methods"
- Chapter 13, "Managing Shared SQL and PL/SQL Areas"
- Chapter 14, "Using Oracle Trace"
- Chapter 15, "Dynamic Performance Views"
- Chapter 16, "Diagnosing System Performance Problems"
- Chapter 17, "Transaction Modes"

11

Overview of Diagnostic Tools

This chapter introduces the full range of diagnostic tools for monitoring production systems and determining performance problems.

This chapter contains the following sections:

- Sources of Data for Tuning
- Dynamic Performance Views
- Oracle and SNMP Support
- EXPLAIN PLAN
- SQL Trace and TKPROF
- Supported Scripts
- Application Registration
- Oracle Enterprise Manager, Packs, and Applications
- Oracle Parallel Server Management
- Independent Tools

Sources of Data for Tuning

This section describes the various sources of data for tuning. Many of these sources may be transient. They include:

- Data Volumes
- Online Data Dictionary
- Operating System Tools
- Dynamic Performance Tables
- Oracle Trace and Oracle Trace Data Viewer
- SQL Trace Facility
- Alert Log
- Application Program Output
- Users
- Initialization Parameter Files
- Program Text
- Design (Analysis) Dictionary
- Comparative Data

Data Volumes

The tuning data source most often overlooked is the data itself. The data may contain information about how many transactions were performed and at what time. The number of rows added to an audit table, for example, can be the best measure of the amount of useful work done; this is known as "the throughput". Where such rows contain a timestamp, you can query the table and use a graphics package to plot throughput against dates and times. Date-stamps and time-stamps need not be apparent to the rest of the application.

If your application does not contain an audit table, be cautious about adding one as it could hinder performance. Consider the trade-off between the value of obtaining the information and the performance cost of doing so.

Online Data Dictionary

The Oracle online data dictionary is a rich source of tuning data when used with the SQL statement `ANALYZE`. This statement stores cluster, table, column, and index statistics within the dictionary, primarily for use by the cost-based optimizer. The dictionary also defines the indexes available to help (or possibly hinder) performance.

Operating System Tools

Tools that gather data at the operating system level are primarily useful for determining scalability, but you should also consult them at an early stage in any tuning activity. In this way you can ensure that no part of the hardware platform is saturated. Network monitors are also required in distributed systems, primarily to check that no network resource is overcommitted. In addition, you can use a simple mechanism such as the UNIX ping command to establish message turnaround time.

See Also: For more information on platform-specific tools, see your operating system documentation.

Dynamic Performance Tables

A number of V$ dynamic performance views are available to help you tune your system and investigate performance problems. They allow you access to memory structures within the SGA.

See Also: For detailed information about each view, see Chapter 15, "Dynamic Performance Views" and *Oracle8i Concepts.*

Oracle Trace and Oracle Trace Data Viewer

Oracle Trace collects Oracle server event activity that includes all SQL and Wait events for specific database users. You can use this information to tune your databases and applications.

See Also: For more information about Oracle Trace and Wait events, see Chapter 14, "Using Oracle Trace".

SQL Trace Facility

SQL trace files record SQL statements issued by a connected process and the resources used by these statements. In general, use V$ views to tune the instance and use SQL trace file output to tune the applications.

See Also: For more information on SQL trace, see "SQL Trace and TKPROF" on page 11-6 and Chapter 6, "Using SQL Trace and TKPROF".

Alert Log

Whenever something unexpected happens in an Oracle environment, check the alert file to see if there is an entry at or around the time of the event.

Application Program Output

In some projects, all application processes (client-side) are instructed to record their own resource consumption to an audit trail. Where database calls are being made through a library, the response time of the client/server mechanism can be inexpensively recorded at the per-call level using an audit trail mechanism. Even without these levels of sophistication, which are not expensive to build or to run, simply preserving resource usages reported by a batch queue manager provides an excellent source of tuning data.

Users

Users normally provide a stream of information as they encounter performance problems.

Initialization Parameter Files

It is vital to have accurate data on exactly what the system was instructed to do and how it was to go about doing it. Some of this data is available from the Oracle parameter files.

Program Text

Data on what the application was to do is also available from the code of the programs or procedures where both the program logic and the SQL statements reside. Server-side code, such as stored procedures, constraints, and triggers, is in this context part of the same data population as client-side code. Tuners must frequently work in situations where the program source code is not available, either as a result of a temporary problem or because the application is a package for which the source code is not released. In such cases it is still important for the tuner to acquire program-to-object cross-reference information. For this reason executable

code is a legitimate data source. Fortunately, SQL is held in text even in executable programs.

Design (Analysis) Dictionary

You can also use the design or analysis dictionary to track intended actions and resource use of the application. Only where the application has been entirely produced by code generators, however, can the design dictionary provide data that would otherwise have to be extracted from programs and procedures.

Comparative Data

Comparative data is invaluable in most tuning situations. Tuning is often conducted from a cold start at each site; the tuners arrive with whatever expertise and experience they may have, plus a few tools for extracting the data. Experienced tuners may recognize similarities in particular situations and attempt to apply a solution that worked elsewhere. Normally, such diagnoses are purely subjective.

Tuning is easier if baselines exist, such as capacity studies performed for this application or data from this or another site running the same application with acceptable performance. The task is then to modify the problematic environment to more closely resemble the optimized environments.

If no directly relevant data can be found, you can check data from similar platforms and similar applications to see if they have the same performance profile. There is no point in trying to tune out a particular effect if it turns out to be ubiquitous.

Dynamic Performance Views

A primary Oracle performance monitoring tool is the dynamic performance views Oracle provides to monitor your system. These view names begin with "V$". This section demonstrates their use in performance tuning. The database user `SYS` owns these views, and administrators can grant any database user access to them. However, only some of these views are relevant to tuning your system.

> **See Also:** For detailed information about each view, see Chapter 15, "Dynamic Performance Views" and *Oracle8i Reference.*

Oracle and SNMP Support

Simple Network Management Protocol (SNMP) enables users to write tools and applications. SNMP is acknowledged as the standard, open protocol for

heterogeneous management applications. Oracle SNMP support enables Oracle databases to be discovered on the network and to be identified and monitored by SNMP-based management applications. Oracle supports several database management information bases (MIBs): the standard MIB for any database management system (independent of vendor), and Oracle-specific MIBs that contain Oracle-specific information. Some statistics mentioned in this manual are supported by these MIBs, and others are not. If you can obtain a statistic mentioned through SNMP, then this fact is noted.

> **See Also:** For more information, see the *Oracle SNMP Support Reference Guide.*

EXPLAIN PLAN

EXPLAIN PLAN is a SQL statement listing the access path used by the query optimizer. Each plan output from the EXPLAIN PLAN statement has a row that provides the statement type.

You should interpret EXPLAIN PLAN results with some discretion. Just because a plan does not seem efficient does not necessarily mean the statement runs slowly. Choose statements for tuning based on their actual resource consumption, not on a subjective view of their execution plans.

> **See Also:** For more information on EXPLAIN PLAN, see Chapter 5, "Using EXPLAIN PLAN" and the *Oracle8i SQL Reference.*

SQL Trace and TKPROF

The SQL trace facility can be enabled for any session. It records in an operating system text file the resource consumption of every parse, execute, fetch, commit, or rollback request made to the server by the session. If the TIMED_STATISTICS parameter is set to true for the session being traced or for the whole system, then this text file also includes the CPU and elapsed time for each statement.

Note: Try to enable SQL trace only for statistics collection, and on specific sessions. If you must enable the facility on an entire production environment, then you can minimize performance impact with the following:

- Maintain at least 25% idle CPU capacity.
- Maintain adequate disk space for the USER_DUMP_DEST location.
- Stripe disk space over sufficient disks.

TKPROF summarizes the trace files produced by the SQL trace facility, optionally including the EXPLAIN PLAN output. TKPROF reports each statement executed with the resources it has consumed, the number of times it was called, and the number of rows it processed. So, it is quite easy to locate individual statements that are using the greatest amount of resources. With experience or with baselines available, you can gauge whether the resources used are reasonable.

See Also: for more information on using SQL trace and TKPROF, see Chapter 6, "Using SQL Trace and TKPROF".

Supported Scripts

Oracle provides many PL/SQL packages, including a good number of SQL*Plus scripts that support instance tuning. Examples include UTLBSTAT.SQL, UTLESTAT.SQL, UTLCHN1.SQL, UTLDTREE.SQL, and UTLLOCKT.SQL. Release 8.1.6 also contains the STATSPACK set of scripts.

These statistical scripts support instance management, allowing you to develop performance history. You can use them to:

- Remove the need to issue DDL each time statistics are gathered.
- Separate data gathering from reporting, and let a range of observations be taken at intervals during a period of representative system operation, and then allow the statistics to be reported from any start point to any end point.
- Report a number of indicative ratios that you can use to determine whether the instance is adequately tuned.
- Present LRU statistics from the buffer cache in a usable form.

STATSPACK differs from the existing UTLBSTAT/UTLESTAT performance scripts in the following ways:

- They collect more data, including high resource SQL.
- Many of the manual calculations which were required with BSTAT/ESTAT are now provided; for example. the first page contains a summary of instance performance and load.
- Permanent tables are created. Each time a new "snapshot" of data is taken, it is added to these tables, with keys which allow comparison between snapshots.
- A new user, PERFSTAT, is automatically created. All objects created by this package are owned by PERFSTAT. This user has limited query-only privileges.
- Written in PL/SQL and uses SQL*Plus as the reporting tool.

Like UTLBSTAT.SQL and UTLESTAT.SQL, STATSPACK can be found in the ORACLE_HOME/rdbms/admin/ directory on UNIX and in the ORACLE_HOME/rdbms81/admin directory on NT.

Application Registration

You can register with the database the name of an application and the actions performed by that application. The application name and actions are recorded in the V$SESSION and V$SQLAREA views. Oracle Trace can also collect application registration data.

Registering an application lets system administrators and tuners track performance by module. System administrators can also use this information to track resource usage by module.

> **See Also:** For more information on registering applications, see *Oracle Enterprise Manager Oracle Trace User's Guide* and *Oracle Enterprise Manager Oracle Trace Developer's Guide*. For more information on the DBMS_APPLICATION_INFO package, see *Oracle8i Supplied PL/SQL Packages Reference*. You can use this package with Oracle Trace and the SQL trace facility to record names of executing modules or transactions in the database for later use when tracking the performance of various modules.

Oracle Enterprise Manager, Packs, and Applications

This section covers:

- Introduction to Oracle Enterprise Manager
- Oracle Diagnostics Pack
  - Oracle Capacity Planner
  - Oracle Performance Manager
  - Oracle Advanced Event Tests
  - Oracle Trace Manager
- Oracle Tuning Pack
  - Oracle Expert
  - Oracle SQL Analyze
  - Oracle Tablespace Manager
  - Oracle Index Tuning Wizard
  - Oracle Auto-Analyze

Introduction to Oracle Enterprise Manager

The Oracle Enterprise Manager (EM) platform is a sophisticated database systems-management environment. This tool provides comprehensive management for Oracle environments.

You can use Enterprise Manager to manage the wide range of Oracle implementations: departmental to enterprise, replication configurations, Web servers, media servers, and so forth. Oracle Enterprise Manager includes:

- A centralized console from which you can run administrative tasks and applications.
- Support to run the Oracle Enterprise Manager console and database administration applications from within a Web browser.
- A lightweight, 3-tier architecture offering unparalleled scalability and failover capability, assuring constant availability of critical management services.
- A centralized repository storing management data for any given environment. Oracle Enterprise Manager supports teams of administrators responsible for cooperatively managing distributed systems.
- Common services for event management, service discovery, and job creation and control.

- Server-side intelligent agent for remote monitoring of events, running jobs, and communicating with the management console.
- Low overhead framework for collecting and managing real-time and historical performance data.
- Applications for administering Oracle databases for security, storage, backup, recovery, import, and software distribution.
- Layered applications for managing replication, Oracle Parallel Server, and other Oracle Server configurations.
- Optional products for monitoring, diagnosing, and planning, known as Oracle Diagnostics Pack.
- Optional products for tuning applications, databases, and systems, known as Oracle Tuning Pack.
- Optional products for managing Oracle metadata changes, known as Oracle Change Management Pack.

The Oracle Enterprise Manager packs provide a set of windows-based and java-based applications built on the Enterprise Manager systems management technology. The Diagnostics Pack and the Tuning Pack are useful in tuning systems and are briefly discussed below.

> **See Also:** For information on the Change Management Pack, see Getting Started with Oracle Change Management Pack

Oracle Diagnostics Pack

The Oracle Diagnostics Pack monitors, diagnoses, and maintains the health of databases, operating systems, and applications. Both historical and real-time analysis are used to automatically avoid problems before they occur. The pack provides powerful capacity planning features enabling you to easily plan and track future system resource requirements.

Oracle Diagnostics Pack components include Oracle Capacity Planner, Oracle Performance Manager, Oracle Advanced Event Tests, Oracle Trace Manager, and Oracle Trace Data Viewer. The following sections describe each component.

Oracle Capacity Planner

Use the Oracle Capacity Planner to collect and analyze historical performance data for your Oracle database and operating system. Oracle Capacity Planner allows you to specify the performance data you want to collect, collection intervals, load

schedules, and data management policies. You can also use Oracle Capacity Planner's in-depth analyses and reports to explore the collected data, to format it into easy-to-use graphs and reports, and to analyze it to predict future resource needs.

Oracle Performance Manager

Oracle Performance Manager captures, computes, and presents performance data for your database and operating system, allowing you to monitor key metrics required to effectively use memory, minimize disk I/O, and to avoid resource contention. It provides a graphical, real-time view of the performance metrics and lets you drill down into a monitoring view for quick access to detailed data for performance problem solving. The performance data is captured and displayed in real-time mode. You can also record the data for replay.

Oracle Performance Manager includes a large set of predefined charts. You can also create your own charts. The graphical monitor is customizable and extensible. You can display monitored information in a variety of two- or three-dimensional graphical views, such as tables, line, bar, cube, and pie charts. You can also customize the monitoring rate.

In addition, Oracle Performance Manager provides a focused view of database activity by database session. The Top Sessions chart extracts and analyzes sample dynamic Oracle performance data by session, automatically determining the top Oracle users based on a specific selection criteria, such as memory consumption, CPU usage, or file I/O activity.

Also, the Database Locks chart within Oracle Performance Manager displays database locks, including details such as the locking user, lock type, object locked, and mode held and requested.

Oracle Advanced Event Tests

Oracle Diagnostics Pack includes Oracle Advanced Event Tests. This is a set of agent-monitored host and database events running on the Oracle Event Management System. You can launch advanced event tests from the console to automatically detect problems on managed servers. Oracle Advanced Event Tests includes predefined events for monitoring database services and system events affecting database performance.

For example, performance-monitoring events include I/O monitoring, memory-structure performance, and user program-response time. I/O monitoring covers disk I/O rates and SQL*Net I/O rates. The tool even allows you to specify an I/O rate threshold; you will receive a warning when this threshold is exceeded.

Memory-structure performance monitoring covers hit rates for the library cache, data dictionary, and database buffers. In addition, you also have the flexibility of monitoring any statistic captured by the dynamic performance table, `V$SYSSTAT`.

You can use Oracle Advanced Event Tests to monitor the status and performance of Oracle storage structures and to detect problems with excessive CPU utilization, excessive CPU load or paging, and disk capacity problems.

In addition to alerting an administrator, Oracle Advanced Event Tests also can be configured to automatically correct the problem event. Using a *Fixit Job*, a predetermined action will automatically occur when an event-alert level is reached.

Oracle Trace Manager

Oracle Trace Manager collects significant Oracle server event data, such as all SQL events and Wait events. SQL events include a complete breakdown of SQL statement activity, such as the parse, execute, and fetch operations. Data collected for server events includes resource usage metrics, such as I/O and CPU consumed by a specific event.

Oracle Trace Data Viewer

Identifying resource-intensive SQL statements is easy with Oracle Trace Data Viewer. The Oracle Trace Data Viewer summarizes Oracle Trace data, including SQL statement metrics such as average elapsed time, CPU consumption, and disk reads per rows fetched.

Oracle Trace collections can be administered through Oracle Trace Manager.

> **See Also:** For more information on Oracle Trace, see Chapter 14, "Using Oracle Trace".

Oracle Tuning Pack

Oracle Tuning Pack optimizes system performance by identifying and tuning major database and application bottlenecks, such as inefficient SQL, poor data structures, and improper use of system resources. The pack proactively discovers tuning opportunities and automatically generates the analysis and required changes to tune the system. Inherent in the product are powerful teaching tools that train DBAs how to tune as they work.

Oracle Expert

Oracle Expert provides automated database performance tuning. Performance problems detected by Oracle Diagnostics Pack and other Oracle monitoring

applications can be analyzed and solved with Oracle Expert. Oracle Expert automates the process of collecting and analyzing data. It contains a rules-based inference engine that provides "expert" database tuning recommendations, implementation scripts, and reports.

Oracle SQL Analyze

Oracle SQL Analyze identifies and helps you tune problematic SQL statements. Use SQL Analyze to detect resource-intensive SQL statements, examine a SQL statement's execution plan, benchmark and compare various optimizer modes and versions of the statement, and generate alternative SQL to improve application performance.

Oracle Tablespace Manager

Oracle Tablespace Manager identifies and corrects Oracle space management problems. Oracle Tablespace Manager has three major features: a Tablespace Allocation graphic, a Tablespace Reorganization tool, and a Tablespace Analyzer tool.

The Tablespace Allocation graphic on the Segments and Extents Information page provides a complete picture of the characteristics of all tablespaces associated with a particular Oracle instance, including tablespace datafiles and segments, total data blocks, free data blocks, and percentage of free blocks available in the tablespace's current storage allocation.

Use the Reorganization tool to rebuild specific objects or an entire tablespace for improved space usage and increased performance. Use the Analyzer tool to automatically keep database statistics up-to-date.

Oracle Index Tuning Wizard

Oracle Index Tuning Wizard automatically identifies tables that would benefit from index changes, determines the best index strategy for each table, presents its findings for verification, and allows you to implement its recommendations.

Oracle Auto-Analyze

Oracle Auto-Analyze maintains your Oracle database statistics. Auto-Analyze runs during a user-specified database maintenance period, thereby reducing adverse performance effects of updating stale statistics. During this maintenance period, Auto-Analyze checks specific schemas for objects that require updating. It also prioritizes the order of objects that require updating and updates the statistics. If the statistics update does not complete during the maintenance period, then

Auto-Analyze maintains the state of the update operation and resumes updating during the next maintenance period.

Oracle Parallel Server Management

Oracle Parallel Server Management is a comprehensive and integrated system management solution for the Oracle Parallel Server. Use Oracle Parallel Server Management to manage multi-instance databases running in heterogeneous environments through an open client-server architecture.

In addition to managing parallel databases, you can use Oracle Parallel Server Management to schedule jobs, perform event management, monitor performance, and obtain statistics to tune parallel databases.

> **See Also:** For more information about Oracle Parallel Server Management, see *Oracle Parallel Server Management Configuration Guide for UNIX* and *Oracle8i Parallel Server Concepts*. For installation instructions, see your platform-specific installation guide.

Independent Tools

At some sites, DBAs have designed in-house performance tools. Such tools might include:

- Free space monitors to determine whether tables have enough space to extend.
- Lock monitoring tools.
- Schema description scripts to show tables and their associated indexes.
- Tools to show default and temporary tablespaces per user.

You can integrate such programs with Oracle by setting them to run automatically.

12

Data Access Methods

This chapter provides an overview of data access methods that can enhance performance, and it warns of situations to avoid. This chapter also explains how to use hints to force various approaches.

This chapter contains the following sections:

- Using Indexes
- Using Function-based Indexes
- Using Bitmap Indexes
- Using Domain Indexes
- Using Clusters
- Using Hash Clusters

Using Indexes

This section describes:

- When to Create Indexes
- Tuning the Logical Structure
- Choosing Columns and Expressions to Index
- Choosing Composite Indexes
- Writing Statements that Use Indexes
- Writing Statements that Avoid Using Indexes
- Assessing the Value of Indexes
- Re-creating Indexes
- Using Nonunique Indexes to Enforce Uniqueness
- Using Enabled Novalidated Constraints

When to Create Indexes

Indexes improve the performance of queries that select a small percentage of rows from a table. As a general guideline, create indexes on tables that are queried for less than 2% or 4% of the table's rows. This value may be higher in situations where all data can be retrieved from an index, or where the indexed columns and expressions can be used for joining to other tables.

This guideline is based on the following assumptions:

- Rows with the same value for the key on which the query is based are uniformly distributed throughout the data blocks allocated to the table.
- Rows in the table are randomly ordered with respect to the key on which the query is based.
- The table contains a relatively small number of columns.
- Most queries on the table have relatively simple `WHERE` clauses.
- The cache hit ratio is low and there is no operating system cache.

If these assumptions do not describe the data in your table and the queries that access it, then an index may only be helpful if your queries typically access at least 25% of the table's rows.

Tuning the Logical Structure

Although cost-based optimization helps avoid the use of nonselective indexes within query execution, the SQL engine must continue to maintain all indexes defined against a table regardless of whether they are used. Index maintenance can present a significant CPU and I/O resource demand in any I/O intensive application. Put another way, building indexes "just in case" is not a good practice; indexes should not be built until required.

To maintain optimal performance with indexes, drop indexes that your application is not using. You can find indexes that are not referenced in execution plans by processing all of your application SQL through `EXPLAIN PLAN` and capturing the resulting plans. Unused indexes are typically, though not necessarily, nonselective.

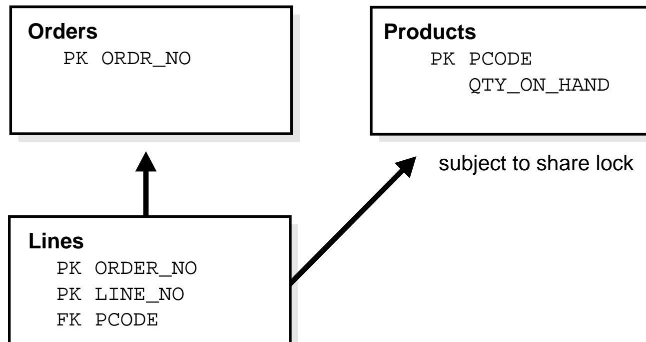
Indexes within an application sometimes have uses that are not immediately apparent from a survey of statement execution plans. In particular, Oracle uses "pins" (nontransactional locks) on foreign key indexes to avoid using shared locks on the child table when enforcing foreign key constraints.

In many applications, a foreign key index never, or rarely, supports a query. In the example shown in Figure 12–1, the need to locate all of the order lines for a given product may never arise. However, when no index exists with `LINES`(`PCODE`) as its leading portion (as described in "Choosing Composite Indexes"), then Oracle places a share lock on the `LINES` table each time `PRODUCTS`(`PCODE`) is updated or deleted. Such a share lock is a problem only if the `PRODUCTS` table is subject to frequent DML.

If this contention arises, then to remove it, the application must either:

- Accept the additional load of maintaining the index.
- Accept the risk of running with the constraint disabled.

Figure 12–1 Foreign Key Constraint



Choosing Columns and Expressions to Index

A key is a column or expression on which you can build an index. Follow these guidelines for choosing index keys to index:

- Consider indexing keys that are frequently used in `WHERE` clauses.
- Consider indexing keys that are frequently used to join tables in SQL statements. For more information on optimizing joins, see the section "Using Hash Clusters" on page 12-26.
- Index keys that have high selectivity. The selectivity of an index is the percentage of rows in a table having the same value for the indexed key. An index's selectivity is optimal if few rows have the same value.

> **Note:** Oracle automatically creates indexes, or uses existing indexes, on the keys and expressions of unique and primary keys that you define with integrity constraints.

You can determine the selectivity of an index by dividing the number of rows in the table by the number of distinct indexed values. You can obtain these values using the `ANALYZE` statement. Selectivity calculated in this manner should be interpreted as a percentage.

Indexes with low selectivity can be helpful if the data distribution is skewed so that one or two values occur much less often than the others. If these values appear frequently in `WHERE` clauses, and if column statistics are gathered so that the optimizer knows which values are rare, then the index can be useful.

- Do not use standard B*-tree indexes on keys or expressions with few distinct values. Such keys or expressions usually have poor selectivity and therefore do not optimize performance unless the frequently selected key values appear less frequently than the other key values. You can use bitmap indexes effectively in such cases, unless a high concurrency OLTP application is involved.
- Do not index columns that are frequently modified. `UPDATE` statements that modify indexed columns and `INSERT` and `DELETE` statements that modify indexed tables take longer than if there were no index. Such SQL statements must modify data in indexes as well as data in tables. They also generate additional undo and redo information.
- Do not index keys that appear only in `WHERE` clauses with functions or operators. A `WHERE` clause that uses a function (other than `MIN` or `MAX`) or an operator with an indexed key does not make available the access path that uses the index.
- Consider indexing foreign keys of referential integrity constraints in cases in which a large number of concurrent `INSERT`, `UPDATE`, and `DELETE` statements access the parent and child tables. Such an index allows `UPDATEs` and `DELETEs` on the parent table without share locking the child table.
- When choosing to index a key, consider whether the performance gain for queries is worth the performance loss for `INSERTs`, `UPDATEs`, and `DELETEs` and the use of the space required to store the index. You may want to experiment by comparing the processing times of your SQL statements with and without indexes. You can measure processing time with the SQL trace facility.

> **See Also:** For more information on the effects of foreign keys on locking, see *Oracle8i Application Developer's Guide - Fundamentals.*

Choosing Composite Indexes

A composite index contains more than one key column. Composite indexes can provide additional advantages over single-column indexes:

Improved selectivity	Sometimes two or more columns or expressions, each with poor selectivity, can be combined to form a composite index with more accurate selectivity.
Reduced I/O	If all columns selected by a query are in a composite index, then Oracle can return these values from the index without accessing the table.

A SQL statement can use an access path involving a composite index if the statement contains constructs that use a leading portion of the index. A leading portion of an index is a set of one or more columns that were specified first and consecutively in the list of columns in the CREATE INDEX statement that created the index. Consider this CREATE INDEX statement:

```
CREATE INDEX comp_ind
ON tab1(x, y, z);
```

These combinations of columns are leading portions of the index: x, xy, and xyz. These combinations of columns are not leading portions of the index: yz, y, and z.

Follow these guidelines for choosing keys for composite indexes:

- Consider creating a composite index on keys that are frequently used together in WHERE clause conditions combined with AND operators, especially if their combined selectivity is better than the selectivity of either key individually.
- If several queries select the same set of keys based on one or more key values, then consider creating a composite index containing all of these keys.

Of course, consider the guidelines associated with the general performance advantages and trade-offs of indexes described in the previous sections. Follow these guidelines for ordering keys in composite indexes:

- Create the index so the keys used in WHERE clauses make up a leading portion.
- If some keys are used in WHERE clauses more frequently, then be sure to create the index so that the more frequently selected keys make up a leading portion to allow the statements that use only these keys to use the index.
- If all keys are used in WHERE clauses equally often, then ordering these keys from most selective to least selective in the CREATE INDEX statement best improves query performance.
- If all keys are used in the WHERE clauses equally often but the data is physically ordered on one of the keys, then place that key first in the composite index.

Writing Statements that Use Indexes

Even after you create an index, the optimizer cannot use an access path that uses the index simply because the index exists. The optimizer can choose such an access path for a SQL statement only if it contains a construct that makes the access path available.

To be sure that a SQL statement can use an access path that uses an index, be sure that the statement contains a construct that makes such an access path available. If you are using the cost-based approach, then also generate statistics for the index. After you have made the access path available for the statement, the optimizer may or may not choose to use the access path, based on the availability of other access paths.

If you create new indexes to tune statements, then you can also use the EXPLAIN PLAN statement to determine whether the optimizer will choose to use these indexes when the application is run. If you create new indexes to tune a statement that is currently parsed, then Oracle invalidates the statement. When the statement is next executed, the optimizer automatically chooses a new execution plan that could potentially use the new index. If you create new indexes on a remote database to tune a distributed statement, then the optimizer considers these indexes when the statement is next parsed.

Also keep in mind that the way you tune one statement may affect the optimizer's choice of execution plans for others. For example, if you create an index to be used by one statement, then the optimizer may choose to use that index for other statements in your application as well. For this reason, you should re-examine your application's performance and rerun the SQL trace facility after you have tuned those statements that you initially identified for tuning.

Writing Statements that Avoid Using Indexes

In some cases, you may want to prevent a SQL statement from using an access path that uses an existing index. You may want to do this if you know that the index is not very selective and that a full table scan would be more efficient. If the statement contains a construct that makes such an index access path available, then you can force the optimizer to use a full table scan through one of these methods:

- You can use the NO_INDEX hint to give the CBO maximum flexibility while disallowing the use of a certain index.
- You can use the FULL hint to force the optimizer to choose a full table scan instead of an index scan.
- You can use the INDEX, INDEX_COMBINE, or AND_EQUAL hints to force the optimizer to use one index or a set of listed indexes instead of another.

Assessing the Value of Indexes

To determine whether an index is good, you must first create it, then analyze it, and use EXPLAIN PLAN on your query to see if the optimizer uses it. If it does, then

keep the index, unless it is expensive to maintain. You can compare the optimizer cost (in the first row of `EXPLAIN PLAN` output) of the plans with and without the index.

Parallel execution uses indexes effectively. It does not perform parallel index range scans, but it does perform parallel index lookups for parallel nested loop join execution. If an index is very selective (there are few rows per index entry), then it may be better to use sequential index lookup than parallel table scan.

Using Fast Full Index Scans

The fast full index scan is an alternative to a full table scan when there is an index that contains all the keys that are needed for the query. A fast full scan is faster than a normal full index scan in that it can use multiblock I/O and can be parallelized just like a table scan. Unlike regular index scans, however, you cannot use keys and the rows will not necessarily come back in sorted order. The following query and plan illustrate this feature.

```
SELECT COUNT(*)
FROM t1, t2
WHERE t1.c1 > 50
    AND t1.c2 = t2.c1;
```

The plan is as follows:

```
SELECT STATEMENT
    SORT AGGREGATE
        HASH JOIN
            TABLE ACCESS t1 FULL
            INDEX t2_c1_idx FAST FULL SCAN
```

Because index `t2_c1_idx` contains all columns needed from table `t2`, the optimizer uses a fast full index scan on that index.

Restrictions

Fast full index scans have the following restrictions:

- At least one indexed column of the table must have the `NOT NULL` constraint.
- Fast full index scans are not possible against bitmap indexes.
- There must be a parallel clause on the index if you want to perform fast full index scan in parallel. The parallel degree of the index is set independently. The index does *not* inherit the degree of parallelism of the table.

- Make sure that you have analyzed the index; otherwise, the optimizer may decide not to use it.

Fast full scan has a special index hint, INDEX_FFS, which has the same format and arguments as the regular INDEX hint.

> **See Also:** For more information on the INDEX_FFS hint, see Chapter 7, "Using Optimizer Hints".

Re-creating Indexes

You may want to re-create an index to compact it and minimize fragmented space, or to change the index's storage characteristics. When creating a new index that is a subset of an existing index, or when rebuilding an existing index with new storage characteristics, Oracle may use the existing index instead of the base table to improve performance.

However, there are cases where it may be beneficial to use the base table instead of the existing index. Consider an index on a table on which a lot of DML has been performed. Because of the DML, the size of the index may increase to the point where each block is only 50% full, or even less. If the index refers to most of the columns in the table, then the index could actually be *larger* than the table. In this case, it is faster to use the base table rather than the index to re-create the index. Another option is to create a new index on a subset of the columns of the original index.

For example, you have a table named cust with columns name, custid, phone, addr, balance, and an index named i_cust_custinfo on table columns name, custid and balance. To create a new index named i_cust_custno on columns custid and name, you would enter:

```
CREATE INDEX i_cust_custno ON cust(custid, name);
```

Oracle automatically uses the existing index (i_cust_custinfo) to create the new index rather than accessing the entire table. The syntax used is the same as if the index i_cust_custinfo did not exist.

Similarly, if you have an index on the empno and mgr columns of the emp table, and if you want to change the storage characteristics of that composite index, then Oracle can use the existing index to create the new index.

Use the ALTER INDEX ... REBUILD statement to reorganize or compact an existing index or to change its storage characteristics. The REBUILD statement uses the existing index as the basis for the new one. All index storage statements are

supported, such as STORAGE (for extent allocation), TABLESPACE (to move the index to a new tablespace), and INITRANS (to change the initial number of entries).

ALTER INDEX ... REBUILD is usually faster than dropping and re-creating an index, because this statement uses the fast full scan feature. It reads all the index blocks using multiblock I/O then discards the branch blocks. A further advantage of this approach is that the old index is still available for queries while the rebuild is in progress.

See Also: For more information about the CREATE INDEX and ALTER INDEX statements, as well as restrictions on re-building indexes, see *Oracle8i SQL Reference.*

Compacting Indexes

You can coalesce leaf blocks of an index using the ALTER INDEX statement with the COALESCE option. This allows you to combine leaf levels of an index to free blocks for re-use. You can also rebuild the index online.

See Also: For more information about the syntax for this statement, see *Oracle8i SQL Reference* and *Oracle8i Administrator's Guide.*

Using Nonunique Indexes to Enforce Uniqueness

You can use an existing nonunique index on a table to enforce uniqueness, either for UNIQUE constraints or the unique aspect of a PRIMARY KEY constraint. The advantage of this approach is that the index remains available and valid when the constraint is disabled. Therefore, enabling a disabled UNIQUE or PRIMARY KEY constraint does not require rebuilding the unique index associated with the constraint. This can yield significant time savings on enable operations for large tables.

Using a nonunique index to enforce uniqueness also lets you eliminate redundant indexes. You do not need a unique index on a primary key column if that column already is included as the prefix of a composite index. You can use the existing index to enable and enforce the constraint. You also save significant space by not duplicating the index. However, if the existing index is partitioned, then the partitioning key of the index must also be a subset of the UNIQUE key; otherwise, Oracle creates an additional unique index to enforce the constraint.

Using Enabled Novalidated Constraints

An enabled novalidated constraint behaves similarly to an enabled validated constraint. Placing a constraint in the enabled novalidated state signifies that any new data entered into the table must conform to the constraint. Existing data is not checked. Placing a constraint in the enabled novalidated state allows you to enable the constraint without locking the table.

If you change a constraint from disabled to enabled, then the table must be locked. No new DML, queries, or DDL can occur because there is no mechanism to ensure that operations on the table conform to the constraint during the enable operation. The enabled novalidated state prevents operations violating the constraint from being performed on the table.

An enabled novalidated constraint can be validated with a parallel, consistent-read query of the table to determine whether any data violates the constraint. No locking is performed and the enable operation does not block readers or writers to the table. In addition, enabled novalidated constraints can be validated in parallel: multiple constraints can be validated at the same time and each constraint's validity check can be determined using parallel query.

Use the following approach to create tables with constraints and indexes:

1. Create the tables with the constraints. `NOT NULL` constraints may be unnamed and should be created enabled and validated. All other constraints (`CHECK`, `UNIQUE`, `PRIMARY KEY`, and `FOREIGN KEY`) should be named and should be "created disabled".

   Note: By default, constraints are created in the `ENABLED` state.

2. Load old data into the tables.
3. Create all indexes including indexes needed for constraints.
4. Enable novalidate all constraints. Do this to primary keys before foreign keys.
5. Allow users to query and modify data.
6. With a separate `ALTER TABLE` statement for each constraint, validate all constraints. Do this to primary keys before foreign keys. For example,

```
CREATE TABLE t (a NUMBER CONSTRAINT apk PRIMARY KEY DISABLE,
b NUMBER NOT NULL);
CREATE TABLE x (c NUMBER CONSTRAINT afk REFERENCES t DISABLE);
```

At this point, use Import or Fast Loader to load data into `t`.

```
CREATE UNIQUE INDEX tai ON t (a);
CREATE INDEX tci ON x (c);
ALTER TABLE t MODIFY CONSTRAINT apk ENABLE NOVALIDATE;
ALTER TABLE x MODIFY CONSTRAINT afk ENABLE NOVALIDATE;
```

Now, users can start performing inserts, updates, deletes, and selects on `t`.

```
ALTER TABLE t ENABLE CONSTRAINT apk;
ALTER TABLE x ENABLE CONSTRAINT afk;
```

Now, the constraints are enabled and validated.

> **See Also:** For a complete discussion of integrity constraints, see *Oracle8i Concepts*.

Using Function-based Indexes

A function-based index is an index on an expression. Oracle strongly recommends using function-based indexes whenever possible. Define function-based indexes anywhere that you use an index on a column, except for columns with `LOBs` or `REFs`. Nested table columns and object types cannot contain these columns.

You can create function-based indexes for any repeatable SQL function. Oracle recommends using function-based indexes for range scans and for functions in `ORDER BY` clauses.

> **Note:** You must set the `QUERY_REWRITE_ENABLED` session parameter to `true` to enable function-based indexes for queries. If `QUERY_REWRITE_ENABLED` is `false`, then function-based indexes are not used for obtaining the values of an expression in the function-based index. However, function-based indexes can still be used for obtaining values in real columns. `QUERY_REWRITE_ENABLED` is a session-level and also an instance-level parameter.

Function-based indexes are an efficient mechanism for evaluating statements that contain functions in `WHERE` clauses. You can create a function-based index to materialize computational-intensive expressions in the index. This permits Oracle to bypass computing the value of the expression when processing `SELECT` and `DELETE` statements. When processing `INSERT` and `UPDATE` statements, however, Oracle evaluates the function to process the statement.

For example, if you create the following index:

```
CREATE INDEX idx ON table_1 (a + b * (c - 1), a, b);
```

Then, Oracle can use it when processing queries such as:

```
SELECT a
FROM table_1
WHERE a + b * (c - 1) < 100;
```

Function-based indexes defined with the `UPPER`(*column_name*) or `LOWER`(*column_name*) keywords allow case-insensitive searches. For example, the following index:

```
CREATE INDEX uppercase_idx ON emp (UPPER(empname));
```

Facilitates processing queries such as:

```
SELECT *
FROM emp
WHERE UPPER(empname) = 'MARK';
```

You can also use function-based indexes for NLS sort indexes that provide efficient linguistic collation in SQL statements.

Oracle treats descending indexes as function-based indexes. The columns marked `DESC` are sorted in descending order.

> **See Also:** For more information on the `CREATE INDEX` statement, see *Oracle8i SQL Reference*.

Function-based Indexes and Index Organized Tables

Use index organized tables (IOTs) on tables with large, non-key columns to speed data retrieval. Because IOTs can store key column values in the indexes and non-key values in the lower leaves of the tree, applications such as those retrieving large text files, coded with a short key value, like an ISBN, might make use of the IOT feature.

The secondary index on an IOT can be a function-based index.

Using Bitmap Indexes

This section describes:

- When to Use Bitmap Indexes

- Creating Bitmap Indexes
- Initialization Parameters for Bitmap Indexing
- Using Bitmap Access Plans on Regular B*-tree Indexes
- Estimating Bitmap Index Size
- Bitmap Index Restrictions

See Also: For more information on bitmap indexing, see *Oracle8i Concepts.*

When to Use Bitmap Indexes

This section describes three aspects of indexing that you must evaluate when deciding whether to use bitmap indexing on a given table:

- Performance Considerations
- Storage Considerations
- Maintenance Considerations

Performance Considerations

Bitmap indexes can substantially improve performance of queries with the following characteristics:

- The `WHERE` clause contains multiple predicates on low- or medium-cardinality columns.
- The individual predicates on these low- or medium-cardinality columns select a large number of rows.
- Bitmap indexes have been created on some or all of these low- or medium-cardinality columns.
- The tables being queried contain many rows.

You can use multiple bitmap indexes to evaluate the conditions on a single table. Bitmap indexes are thus highly advantageous for complex ad hoc queries that contain lengthy `WHERE` clauses. Bitmap indexes can also provide optimal performance for aggregate queries and for optimizing joins in star schemas.

See Also: For more information on optimizing anti-joins and semi-joins, see *Oracle8i Concepts.*

Storage Considerations

Bitmap indexes can provide considerable storage savings over the use of B*-tree indexes. In databases containing only B*-tree indexes, you must anticipate the columns that would commonly be accessed together in a single query, and create a composite B*-tree index on these columns.

Not only would this B*-tree index require a large amount of space, it would also be ordered. That is, a B*-tree index on (`marital_status`, `region`, `gender`) is useless for queries that only access `region` and `gender`. To completely index the database, you must create indexes on the other permutations of these columns. For the simple case of three low-cardinality columns, there are six possible composite B*-tree indexes. You must consider the trade-offs between disk space and performance needs when determining which composite B*-tree indexes to create.

Bitmap indexes solve this dilemma. Bitmap indexes can be efficiently combined during query execution, so three small single-column bitmap indexes can do the job of six three-column B*-tree indexes.

Bitmap indexes are much more efficient than B*-tree indexes, especially in data warehousing environments. Bitmap indexes are created not only for efficient space usage, but also for efficient execution, and the latter is somewhat more important.

Do not create bitmap indexes on unique key columns. However, for columns where each value is repeated hundreds or thousands of times, a bitmap index typically is less than 25% of the size of a regular B*-tree index. The bitmaps themselves are stored in compressed format.

Simply comparing the relative sizes of B*-tree and bitmap indexes is not an accurate measure of effectiveness, however. Because of their different performance characteristics, you should keep B*-tree indexes on high-cardinality columns, while creating bitmap indexes on low-cardinality columns.

Maintenance Considerations

Bitmap indexes benefit data warehousing applications, but they are not appropriate for OLTP applications with a heavy load of concurrent `INSERTs`, `UPDATEs`, and `DELETEs`. In a data warehousing environment, data is usually maintained by way of bulk inserts and updates. Index maintenance is deferred until the end of each DML operation. For example, if you insert 1000 rows, then the inserted rows are placed into a sort buffer, and then the updates of all 1000 index entries are batched.

(This is why SORT_AREA_SIZE must be set properly for good performance with inserts and updates on bitmap indexes.) Thus, each bitmap segment is updated only once per DML operation, even if more than one row in that segment changes.

Note: The sorts described above are regular sorts and use the regular sort area, determined by SORT_AREA_SIZE. The BITMAP_MERGE_AREA_SIZE and CREATE_BITMAP_AREA_SIZE parameters described in "Initialization Parameters for Bitmap Indexing" on page 12-19 only affect the specific operations indicated by the parameter names.

DML and DDL statements, such as UPDATE, DELETE, DROP TABLE, affect bitmap indexes the same way they do traditional indexes: the consistency model is the same. A compressed bitmap for a key value is made up of one or more bitmap segments, each of which is at most half a block in size (but may be smaller). The locking granularity is one such bitmap segment. This may affect performance in environments where many transactions make simultaneous updates. If numerous DML operations have caused increased index size and decreasing performance for queries, then you can use the ALTER INDEX ... REBUILD statement to compact the index and restore efficient performance.

A B*-tree index entry contains a single rowid. Therefore, when the index entry is locked, a single row is locked. With bitmap indexes, an entry can potentially contain a range of rowids. When a bitmap index entry is locked, the entire range of rowids is locked. The number of rowids in this range affects concurrency. As the number of rowids increases in a bitmap segment, concurrency decreases.

Locking issues affect DML operations, and may affect heavy OLTP environments. Locking issues do not, however, affect query performance. As with other types of indexes, updating bitmap indexes is a costly operation. Nonetheless, for bulk inserts and updates where many rows are inserted or many updates are made in a single statement, performance with bitmap indexes can be better than with regular B*-tree indexes.

Creating Bitmap Indexes

To create a bitmap index, use the BITMAP keyword in the CREATE INDEX statement:

```
CREATE BITMAP INDEX ...
```

Multi-column (concatenated) bitmap indexes are supported. They can be defined over no more than 30 columns. Other SQL statements concerning indexes, such as DROP, ANALYZE, ALTER, and so on, can refer to bitmap indexes without any extra keyword.

See Also: For information on bitmap index restrictions, see *Oracle8i SQL Reference.*

Index Type

System index views USER_INDEXES, ALL_INDEXES, and DBA_INDEXES indicate bitmap indexes by the word BITMAP appearing in the TYPE column. A bitmap index cannot be declared as UNIQUE. A bitmap index on a unique key is useless.

Using Hints

The INDEX hint works with bitmap indexes in the same way as with traditional indexes.

The INDEX_COMBINE hint identifies the most cost effective indexes for the optimizer. The optimizer recognizes all indexes that can potentially be combined, given the predicates in the WHERE clause. However, it may not be cost effective to use all of them. Oracle recommends using INDEX_COMBINE rather than INDEX for bitmap indexes, because it is a more versatile hint.

In deciding which of these hints to use, the optimizer includes non-hinted indexes that appear cost effective, as well as indexes named in the hint. If certain indexes are given as arguments for the hint, then the optimizer tries to use some combination of those particular bitmap indexes.

If the hint does not name indexes, then all indexes are considered hinted. Hence, the optimizer tries to combine as many as is possible given the WHERE clause, without regard to cost effectiveness. The optimizer always tries to use hinted indexes in the plan regardless of whether it considers them cost effective.

See Also: For more information on the INDEX_COMBINE hint, see Chapter 7, "Using Optimizer Hints".

Performance and Storage Tips

To get optimal performance and disk space usage with bitmap indexes, consider the following tips:

- Large block sizes improve the efficiency of storing, and hence, retrieving, bitmap indexes.

- To make compressed bitmaps as small as possible, declare NOT NULL constraints on all columns that cannot contain null values.
- Fixed-length datatypes are more amenable to a compact bitmap representation than variable length datatypes.

This is because Oracle needs to consider the theoretical maximum number of rows that will fit in a data block when creating bitmap indexes.

> **See Also:** For more information about bitmap EXPLAIN PLAN output, see Chapter 5, "Using EXPLAIN PLAN"

Efficient Mapping of Bitmaps to Rowids

Use SQL statements with the ALTER TABLE syntax to optimize the mapping of bitmaps to rowids. The MINIMIZE RECORDS_PER_BLOCK clause enables this optimization and the NOMINIMIZE RECORDS_PER_BLOCK clause disables it.

When enabled, Oracle scans the table and determines the maximum number of records in any block and restricts this table to this maximum number. This enables bitmap indexes to allocate fewer bits per block and results in smaller bitmap indexes. The block and record allocation restrictions this statement places on the table are only beneficial to bitmap indexes. Therefore, Oracle does not recommend using this mapping on tables that are not heavily indexed with bitmap indexes.

> **See Also:** For more information, see "Using Bitmap Indexes" on page 12-13. For more information on MINIMIZE and NOMINIMIZE syntax, see *Oracle8i SQL Reference.*

Indexing Null Values

Bitmap indexes index nulls, whereas all other index types do not. Consider, for example, a table with STATE and PARTY columns, on which you want to perform the following query:

```
SELECT COUNT(*)
FROM people
WHERE state='CA'
    AND party !='D';
```

Indexing nulls enables a bitmap minus plan where bitmaps for party equal to D and NULL are subtracted from state bitmaps equal to CA. The EXPLAIN PLAN output would look like this:

```
SELECT STATEMENT
  SORT AGGREGATE
```

```
BITMAP CONVERSION COUNT
  BITMAP MINUS
    BITMAP MINUS
      BITMAP INDEX SINGLE VALUE STATE_BM
      BITMAP INDEX SINGLE VALUE PARTY_BM
    BITMAP INDEX SINGLE VALUE PARTY_BM
```

If a NOT NULL constraint existed on party, then the second minus operation (where party is null) would be left out because it is not needed.

Initialization Parameters for Bitmap Indexing

The following initialization parameters have an effect on performance:

- CREATE_BITMAP_AREA_SIZE
- BITMAP_MERGE_AREA_SIZE
- SORT_AREA_SIZE

CREATE_BITMAP_AREA_SIZE

This parameter determines the amount of memory allocated for bitmap creation. The default value is 8MB. A larger value may lead to faster index creation. If cardinality is very small, then you can set a small value for this parameter. For example, if cardinality is only 2, then the value can be on the order of kilobytes rather than megabytes. As a general rule, the higher the cardinality, the more memory is needed for optimal performance. You cannot dynamically alter this parameter at the system or session level.

BITMAP_MERGE_AREA_SIZE

This parameter determines the amount of memory used to merge bitmaps retrieved from a range scan of the index. The default value is 1 MB. A larger value should improve performance because the bitmap segments must be sorted before being merged into a single bitmap. You cannot dynamically alter this parameter at the system or session level.

SORT_AREA_SIZE

This parameter must be set properly for good performance with inserts and updates on bitmap indexes. Thus, each bitmap segment is updated only once per DML operation, even if more than one row in that segment changes.

See Also: For more information on improving bitmap index efficiency, see "Efficient Mapping of Bitmaps to Rowids" on page 12-18.

Using Bitmap Access Plans on Regular B*-tree Indexes

If there is at least one bitmap index on the table, then the optimizer considers using a bitmap access path using regular B*-tree indexes for that table. This access path may involve combinations of B*-tree and bitmap indexes, but may not involve any bitmap indexes at all. However, the optimizer will not generate a bitmap access path using a single B*-tree index unless instructed to do so by a hint.

To use bitmap access paths for B*-tree indexes, the rowids stored in the indexes must be converted to bitmaps. After such a conversion, the various Boolean operations available for bitmaps can be used. As an example, consider the following query, where there is a bitmap index on column `c1`, and regular B*-tree indexes on columns `c2` and `c3`.

```
EXPLAIN PLAN FOR
SELECT COUNT(*)
FROM t
WHERE c1 = 2 AND c2 = 6
OR c3 BETWEEN 10 AND 20;

SELECT STATEMENT
     SORT AGGREGATE
       BITMAP CONVERSION COUNT
         BITMAP OR
           BITMAP AND
             BITMAP INDEX c1_ind SINGLE VALUE
             BITMAP CONVERSION FROM ROWIDS
               INDEX c2_ind RANGE SCAN
           BITMAP CONVERSION FROM ROWIDS
             SORT ORDER BY
               INDEX c3_ind RANGE SCAN
```

Note: This statement is executed by accessing indexes only, so no table access is necessary.

Here, a `COUNT` option for the `BITMAP CONVERSION` row source counts the number of rows matching the query. There are also conversions `FROM` rowids in the plan to generate bitmaps from the rowids retrieved from the B*-tree indexes. The occurrence of the `ORDER BY` sort in the plan is due to the fact that the conditions on

column c3 result in more than one list of rowids being returned from the B*-tree index. These lists are sorted before they can be converted into a bitmap.

Estimating Bitmap Index Size

Although it is not possible to precisely size a bitmap index, you can *estimate* its size. This section describes how to determine the size of a bitmap index for a table using the computed size of a B*-tree index. It also illustrates how cardinality, NOT NULL constraints, and the number of distinct values affect bitmap size.

To estimate the size of a bitmap index for a given table, extrapolate the size of a B*-tree index for the table. Use the following approach:

1. Use the standard formula described in *Oracle8i Concepts* to compute the size of a B*-tree index for the table.
2. Determine the cardinality of the table data.
3. From the cardinality value, extrapolate the size of a bitmap index, according to the graph in Figure 12–2 or Figure 12–3.

For a 1 million row table, Figure 12–2 shows index size on columns with different numbers of distinct values for B*-tree indexes and bitmap indexes. Using Figure 12–2, you can estimate the size of a bitmap index relative to that of a B*-tree index for the table. Sizing is not exact: results vary somewhat from table to table.

Randomly distributed data was used to generate the graph. If, in your data, particular values tend to cluster close together, then you may generate considerably smaller bitmap indexes than indicated by the graph. Also, bitmap indexes may be slightly smaller than those in the graph if columns contain NOT NULL constraints.

Figure 12–3 shows similar data for a table with 5 million rows. When cardinality exceeds 100,000, bitmap index size does not increase as fast as it does in Figure 12–2. For a table with more rows, there are more repeating values for a given cardinality.

Figure 12–2 Extrapolating Bitmap Index Size: 1 Million Row Table

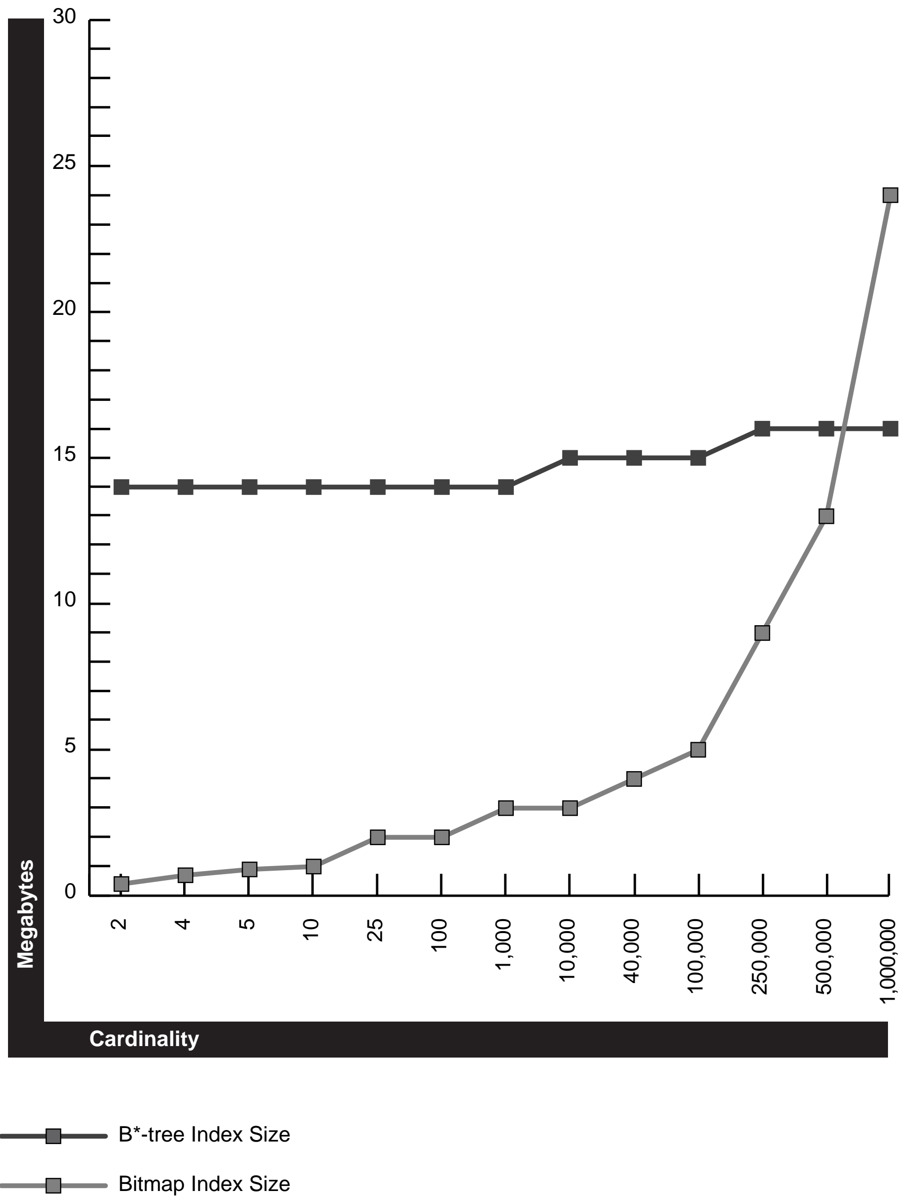
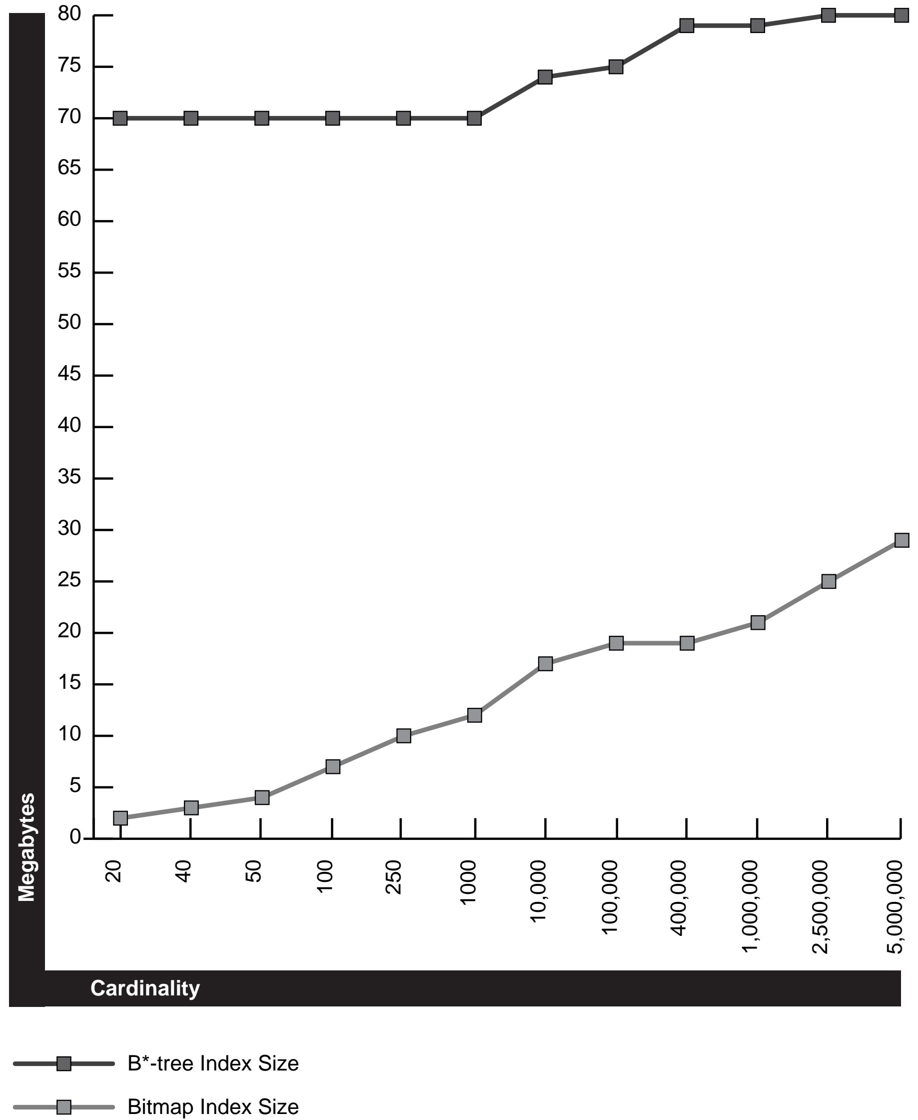


Figure 12–3 Extrapolating Bitmap Index Size: 5 Million Row Table



Bitmap Index Restrictions

Bitmap indexes have the following restrictions:

- For bitmap indexes with direct load, the `SORTED_INDEX` flag does not apply.
- Bitmap indexes are not considered by the rule-based optimizer.
- Bitmap indexes cannot be used for referential integrity checking.

Using Domain Indexes

Domain indexes are built using the indexing logic supplied by a user-defined indextype. An indextype provides an efficient mechanism to access data that satisfy certain operator predicates. Typically, the user-defined indextype is part of an Oracle option, like the Spatial option.

For example, the `SpatialIndextype` allows efficient search and retrieval of spatial data that overlap a given bounding box.

The cartridge determines the parameters you can specify in creating and maintaining the domain index. Similarly, the performance and storage characteristics of the domain index are presented in the specific cartridge documentation.

Refer to the appropriate cartridge documentation for information such as:

- What datatypes can be indexed?
- What indextypes are provided?
- What operators does the indextype support?
- How can the domain index be created and maintained?
- How do we efficiently use the operator in queries?
- What are the performance characteristics?

Note: You can also create index types with the `CREATE INDEXTYPE` SQL statement.

See Also: For information about the `SpatialIndextype`, see the *Oracle Spatial User's Guide and Reference*.

Using Clusters

A cluster is a group of tables that share the same data blocks because they share common columns and are often used together.

> **See Also:** For more information on clusters, see *Oracle8i Concepts*.

Follow these guidelines when deciding whether to cluster tables:

- Cluster tables that are often accessed by your application in join statements.
- Do not cluster tables if your application joins them only occasionally or modifies their common column values frequently. Modifying a row's cluster key value takes longer than modifying the value in an unclustered table, because Oracle may have to migrate the modified row to another block to maintain the cluster.
- Do not cluster tables if your application often performs full table scans of only one of the tables. A full table scan of a clustered table can take longer than a full table scan of an unclustered table. Oracle is likely to read more blocks because the tables are stored together.
- Cluster master-detail tables if you often select a master record and then the corresponding detail records. Detail records are stored in the same data block(s) as the master record, so they are likely still to be in memory when you select them, requiring Oracle to perform less I/O.
- Store a detail table alone in a cluster if you often select many detail records of the same master. This measure improves the performance of queries that select detail records of the same master but does not decrease the performance of a full table scan on the master table.
- Do not cluster tables if the data from all tables with the same cluster key value exceeds more than one or two Oracle blocks. To access a row in a clustered table, Oracle reads all blocks containing rows with that value. If these rows take up multiple blocks, then accessing a single row could require more reads than accessing the same row in an unclustered table.
- Do not cluster tables when the number of rows per cluster key value varies significantly. This causes space wastage for the low cardinality key value and collisions for the high cardinality key values. Collisions degrade performance.

Consider the benefits and drawbacks of clusters with respect to the needs of your application. For example, you may decide that the performance gain for join statements outweighs the performance loss for statements that modify cluster key

values. You may want to experiment and compare processing times with your tables both clustered and stored separately. To create a cluster, use the `CREATE CLUSTER` statement.

> **See Also:** For more information on creating clusters, see *Oracle8i Application Developer's Guide - Fundamentals.*

Using Hash Clusters

Hash clusters group table data by applying a hash function to each row's cluster key value. All rows with the same cluster key value are stored together on disk. Consider the benefits and drawbacks of hash clusters with respect to the needs of your application. You may want to experiment and compare processing times with a particular table as it is stored in a hash cluster, and as it is stored alone with an index. This section describes:

- When to Use Hash Clusters
- Creating Hash Clusters

When to Use Hash Clusters

Follow these guidelines for choosing when to use hash clusters:

- Use hash clusters to store tables often accessed by SQL statements with `WHERE` clauses if the `WHERE` clauses contain equality conditions that use the same column or combination of columns. Designate this column or combination of columns as the cluster key.
- Store a table in a hash cluster if you can determine how much space is required to hold all rows with a given cluster key value, including rows to be inserted immediately as well as rows to be inserted in the future.
- Do not use hash clusters if space in your database is scarce, and you cannot afford to allocate additional space for rows to be inserted in the future.
- Do not use a hash cluster to store a constantly growing table if the process of occasionally creating a new, larger hash cluster to hold that table is impractical.
- Do not store a table in a hash cluster if your application often performs full table scans, and you must allocate a great deal of space to the hash cluster in anticipation of the table growing. Such full table scans must read all blocks allocated to the hash cluster, even though some blocks may contain few rows. Storing the table alone would reduce the number of blocks read by full table scans.

- Do not store a table in a hash cluster if your application frequently modifies the cluster key values. Modifying a row's cluster key value can take longer than modifying the value in an unclustered table, because Oracle may have to migrate the modified row to another block to maintain the cluster.
- Storing a single table in a hash cluster can be useful, regardless of whether the table is often joined with other tables, provided that hashing is appropriate for the table based on the previous points in this list.

Creating Hash Clusters

To create a hash cluster, use the `CREATE CLUSTER` statement with the `HASHKEYS` parameter.

When you create a hash cluster, you must use the `HASHKEYS` parameter of the `CREATE CLUSTER` statement to specify the number of hash values for the hash cluster. For best performance of hash scans, choose a `HASHKEYS` value that is at least as large as the number of cluster key values. Such a value reduces the chance of collisions, or multiple cluster key values resulting in the same hash value. Collisions force Oracle to test the rows in each block for the correct cluster key value after performing a hash scan. Collisions reduce the performance of hash scans.

Oracle always rounds up the `HASHKEYS` value that you specify to the nearest prime number to obtain the actual number of hash values. This rounding is designed to reduce collisions.

> **See Also:** For more information on creating hash clusters, see *Oracle8i Application Developer's Guide - Fundamentals.*

13

Managing Shared SQL and PL/SQL Areas

Oracle compares SQL statements and PL/SQL blocks issued directly by users and applications, as well as recursive SQL statements issued internally by a DDL statement. If two exact statements are issued, then the SQL or PL/SQL area used to process the first instance of the statement is *shared*. This means that it is used for the processing of the subsequent executions of that same statement. *Similar* statements also share SQL areas when the `CURSOR_SHARING` parameter is set to `FORCE`.

> **See Also:** For more information on similar SQL statements, see Chapter 19, "Tuning Memory Allocation".

Shared SQL and PL/SQL areas are shared memory areas. Any Oracle process can use a shared SQL area. Shared SQL areas reduce memory usage on the database server, thereby increasing system throughput. Shared SQL and PL/SQL areas age out of the shared pool according to a "least recently used" (LRU) algorithm, similar to database buffers. To improve performance and prevent reparsing, you may want to prevent large SQL or PL/SQL areas from aging out of the shared pool.

> **Note:** Shared SQL is not recommended with data warehousing applications. Use literal values in these SQL statements, rather than bind variables. If you use bind variables, then the optimizer makes a blanket assumption about the selectivity of the column. However, if you specify a literal value, then the optimizer can use value histograms and provide a better access plan.

This chapter contains the following sections:

- Comparing SQL Statements and PL/SQL Blocks
- Keeping Shared SQL and PL/SQL in the Shared Pool

Comparing SQL Statements and PL/SQL Blocks

This section describes the following:

- Testing for Identical SQL Statements
- Aspects of Standardized SQL Formatting

Testing for Identical SQL Statements

Oracle automatically notices when two or more applications send identical SQL statements or PL/SQL blocks to the database. It does not need to parse a statement to determine whether it is identical to another statement currently in the shared pool. Oracle distinguishes identical statements using the following steps:

1. The text string of an issued statement is hashed. If the hash value is the same as a hash value for an existing SQL statement in the shared pool, then Oracle proceeds to Step 2.
2. The text string of the issued statement, including case, blanks, and comments, is compared to all existing SQL statements that were identified in Step 1.
3. The objects referenced in the issued statement are compared to the referenced objects of all existing statements identified in Step 2. For example, if two users each have `emp` tables, then the statement

   ```
   SELECT * FROM emp;
   ```

   is not considered identical, because the statement references different tables for each user.
4. The bind types of bind variables used in a SQL statement must match.

Note: Most Oracle products convert the SQL before passing statements to the database. Characters are uniformly changed to upper case, white space is compressed, and bind variables are renamed so that a consistent set of SQL statements is produced.

Aspects of Standardized SQL Formatting

It is neither necessary nor useful to have every user of an application attempt to write SQL statements in a standardized way. It is unlikely that 300 people writing ad hoc dynamic statements in standardized SQL generate the same SQL statements. The chances that they all want to look at exactly the same columns, in exactly the same tables, in exactly the same order is remote. By contrast, 300 people running the same application—executing command files—*will* generate the same SQL statements.

Within an application, there is a very minimal advantage to having 300 users use two identical statements; however, there is a major advantage to having one statement used by 600 users.

Keeping Shared SQL and PL/SQL in the Shared Pool

This section describes two techniques of keeping shared SQL and PL/SQL in the shared pool:

- Reserving Space for Large Allocations
- Preventing Objects from Aging Out

Reserving Space for Large Allocations

A problem can occur if users fill the shared pool, and then a large package ages out. If someone calls the large package back in, then a significant amount of maintenance is required to create space for it in the shared pool. You can avoid this problem by reserving space for large allocations with the SHARED_POOL_RESERVED_SIZE initialization parameter. This parameter sets aside room in the shared pool for allocations larger than the value specified by the SHARED_POOL_RESERVED_SIZE_MIN_ALLOC parameter.

Note: Although Oracle uses segmented codes to reduce the need for large areas of contiguous memory, performance may improve if you pin large objects in memory.

See Also: For more information on the SHARED_POOL_RESERVED_SIZE parameter, see "Tuning the Shared Pool" in Chapter 19, "Tuning Memory Allocation".

Preventing Objects from Aging Out

The DBMS_SHARED_POOL package lets you keep objects in shared memory, so that they do not age out with the normal LRU mechanism. By using the DBMS_SHARED_POOL package, and by loading the SQL and PL/SQL areas before memory fragmentation occurs, the objects can be kept in memory. This ensures that memory is available, and it prevents the sudden, inexplicable slowdowns in user response time that occur when SQL and PL/SQL areas are accessed after aging out.

> **See Also:** For more information on using DBMS_SHARED_POOL, see *Oracle8i Supplied PL/SQL Packages Reference.*

When to Use DBMS_SHARED_POOL

- The procedures provided with the DBMS_SHARED_POOL package may be useful when loading large PL/SQL objects, such as the STANDARD and DIUTIL packages. When large PL/SQL objects are loaded, user response time is affected. This is because of the large number of smaller objects that need to age out of the shared pool to make room (due to memory fragmentation). In some cases, there may be insufficient memory to load the large objects.
- DBMS_SHARED_POOL is useful for frequently executed triggers. You may want to keep compiled triggers on frequently used tables in the shared pool.
- DBMS_SHARED_POOL also supports sequences. Sequence numbers are lost when a sequence ages out of the shared pool. DBMS_SHARED_POOL keeps sequences in the shared pool, thus preventing the loss of sequence numbers.

How to Use DBMS_SHARED_POOL

To use the DBMS_SHARED_POOL package to pin a SQL or PL/SQL area, complete the following steps.

1. Decide which packages or cursors to pin in memory.
2. Start up the database.
3. Make the call to DBMS_SHARED_POOL.KEEP to pin your objects.

   This procedure ensures that your system does not run out of shared memory before the objects are loaded. By pinning the objects early in the life of the instance, you prevent memory fragmentation that could result from pinning a large portion of memory in the middle of the shared pool.

> **See Also:** For specific information on using DBMS_SHARED_POOL procedures, see *Oracle8i Supplied PL/SQL Packages Reference.*

14

Using Oracle Trace

This chapter describes how to use Oracle Trace to collect Oracle server event data.

This chapter contains the following sections:

- Introduction to Oracle Trace
- Using Oracle Trace Manager
- Using Oracle Trace Data Viewer
- Manually Collecting Oracle Trace Data

Introduction to Oracle Trace

Oracle Trace is a general-purpose data collection product and is part of the Oracle Enterprise Manager systems management product family. The Oracle server uses Oracle Trace to collect performance and resource utilization data, such as SQL Parse, Execute, Fetch statistics, and Wait statistics.

> **See Also:** For more information, see *Oracle Enterprise Manager Oracle Trace User's Guide* and *Oracle Enterprise Manager Oracle Trace Developer's Guide*. These books contain a complete list of events and data that you can collect for the Oracle server, as well as information on how to implement tracing in your own products and applications.

Using Oracle Trace Data

Among the many advantages of using Oracle Trace is the integration of Oracle Trace with many other applications. You can use Oracle Trace data collected for the Oracle server in the following applications, as shown in Figure 14–1:

- Oracle Expert

  You can use information collected with Oracle Trace as an optional source of SQL workload data in Oracle Expert. This SQL data is used when recommending the addition or removal of indexes.

  > **See Also:** For more information, see *Database Tuning with the Oracle Tuning Pack*.

- Oracle Trace Data Viewer

  Oracle Trace Data Viewer is a simple viewer for inspecting Oracle Trace collections containing SQL and Wait statistics. You can export Oracle Trace Data to the following products for further analysis:

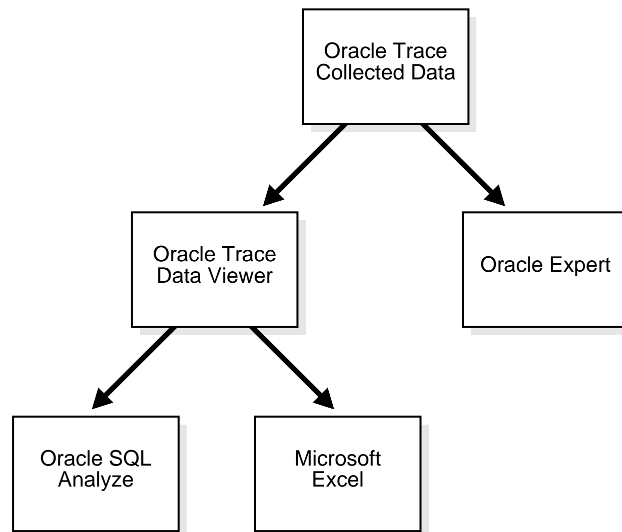
  – SQL Analyze

    You can select one or more rows in Data Viewer and save the SQL statement text to a file that you can import into SQL Analyze. You can then use SQL Analyze to tune these individual statements.

  – Third-Party Tools, such as Microsoft Excel

    SQL in Data Viewer can be saved to a CSV (Comma Separated Value) file for viewing in third-party tools, such as Microsoft Excel.

Figure 14–1 Integration of Oracle Trace with Other Applications



Importing Oracle Trace Data into Oracle Expert

You can use Oracle Trace to collect workload data for use in the Oracle Expert application. Oracle Trace collects resource utilization statistics for SQL statements executing against a database in real time. Oracle Trace allows you to collect data about all the SQL statements executing against a database *during* periods of poor performance.

You control the scheduling and duration of an Oracle Trace collection period. To obtain SQL workload data for a 15-minute period of poor performance, stop collection immediately after the poor performance interval ends.

Importing Data Viewer SQL Into Oracle SQL Analyze

While using Data Viewer, you can select one or more rows in the top portion of the Data View window to save to a file. When you choose SQL (SQL Analyze Format) from File/Save, a file containing query text is saved. You can then import this `*.sql` file into Oracle SQL Analyze for tuning of the selected statements.

Oracle SQL Analyze can show you the execution plan for individual queries and let you experiment with various optimizer modes and hints.

Importing Data Viewer Information into Third-Party Tools

While using Data Viewer, you can select one or more rows in the top portion of the Data View window to save to a file. When you choose the CSV file format, Oracle Trace creates a `*.csv` file that you can load into a third-party tool, such as a Microsoft Excel spreadsheet.

Using Oracle Trace Manager

Oracle Trace provides a graphical Oracle Trace Manager application to create, schedule, and administer Oracle Trace collections for products containing Oracle Trace calls.

The Oracle server has been coded with Oracle Trace API calls to collect both SQL and Wait statistics with a minimum of overhead. Using the Oracle Trace Manager graphical user interface you can:

- Schedule collections.
- Filter collections by user.
- Filter collections by type of Wait event.
- Format collected data to database tables to preserve historical data.
- View SQL and Wait statistics using Oracle Trace Data Viewer.

Managing Collections

Use and control of Oracle Trace revolves around the concept of a "collection." A collection is data collected for events that occurred while a product with Oracle Trace code was running.

With the Oracle Trace Manager, you can schedule and manage collections. When creating a collection, you define the attributes of the collection, such as the collection name, the data to be included in the collection, and the start and end times. The Oracle Trace Manager includes a Collection Wizard that facilitates the creation and execution of collections.

After you create a collection you can execute it immediately or schedule it to execute at a specific time or at specified intervals. When a collection executes, it produces a file containing the data for the products participating in the collection. You can also use a collection as a template for creating other similar collections.

Collecting Event Data

An event is the occurrence of some activity within a product. Oracle Trace collects data for predefined events occurring within a software product created with the Oracle Trace API. That is, the product is embedded with Oracle Trace API calls. An example of an event is a parse or fetch.

There are two types of events:

- Point events

  Point events represent an instantaneous occurrence of something in the instrumented product. An example of a point event is an error occurrence.

- Duration events

  Duration events have a beginning and ending. An example of a duration event is a transaction. Duration events can have other events occur within them; for example, an error can occur *within* a transaction.

The Oracle server is instrumented for 13 events. Three of these events are:

- Database Connection: A point event that records data such as the server login user name.
- SQL Parse: One of the series of SQL processing duration events. This event records a large set of data such as sorts, resource use, and cursor numbers.
- RowSource: Data about the execution plan, such as SQL operation, position, object identification, and number of rows processed by a single row source within an execution plan.

Accessing Collected Data

During a collection, Oracle Trace buffers event data in memory and periodically writes it to a collection binary file. This method ensures low overhead associated with the collection process. You can access event data collected in the binary file by formatting the data to predefined tables which makes the data available for fast, flexible access. These predefined tables are called "Oracle Trace formatter tables."

Oracle Trace Manager provides a mechanism for formatting collection data immediately after a collection or at a later time.

When formatting a collection, you identify the database where Oracle Trace Manager creates the formatted collection as follows:

1. Using Oracle Trace Manager, select a collection to format.

2. Choose the Format statement.
3. Specify a target database where the data is to reside.

The collection you select determines which collection definition file and data collection file is used. The formatted target database determines where the formatted collection data is stored.

After the data is formatted, you can access the data using the Oracle Trace Data Viewer or by using SQL reporting tools and scripts.

Also, you can access event data by running the Detail report from the Oracle Trace reporting utility. This report provides a basic mechanism for viewing a collection's results. You have limited control over what data is reported and how it is presented.

> **See Also:** For more information about predefined SQL scripts and the Detail reports, see *Oracle Enterprise Manager Oracle Trace Developer's Guide.*

Using Oracle Trace Data Viewer

After using Oracle Trace to collect data, run the Data Viewer by selecting "View Formatted Data..." from the Oracle Trace Collection menu. Or you can select it directly from the Oracle Diagnostics Pack toolbar. Data Viewer can compute SQL and Wait statistics and resource utilization metrics from the raw data that is collected. After Data Viewer computes statistics, targeting resource intensive SQL becomes a much simpler task.

Data Viewer computes SQL statistics from data collected by Oracle Trace Manager for all executions of a query during the collection period. Resource utilization during a single execution of a SQL statement may be misleading due to other concurrent activities on the database or node. Combining statistics for all executions may lend a clearer picture about the typical resource utilization occurring when a given query is executed.

> **Note:** You can filter out SQL statements executed by `SYS`.

Oracle Trace Predefined Data Views

SQL and Wait statistics are presented in a comprehensive set of Oracle Trace predefined data views. Within Wait statistics, a data view is the definition of a query

into the data collected by Oracle Trace. A data view consists of items or statistics to be returned and optionally a sort order and limit of rows to be returned.

With the data views provided by Data Viewer, you can:

- Examine important statistical data, for example, elapsed times or disk-to-logical-read hit rates.
- Drill down as needed to get additional details about the statement's execution.

In addition to the predefined data views, you can define your own data views using the Create Data View Wizard.

After Data Viewer has computed SQL and Wait statistics, a dialog box showing the available data views appears. SQL Statistic data views are grouped by I/O, Parse, Elapsed Time, CPU, Row, Sort, and Wait statistics as shown in Figure 14–2. A description of the selected data view is shown on the right-hand side of the screen.

Figure 14–2 Oracle Trace Data Viewer - Collection Screen

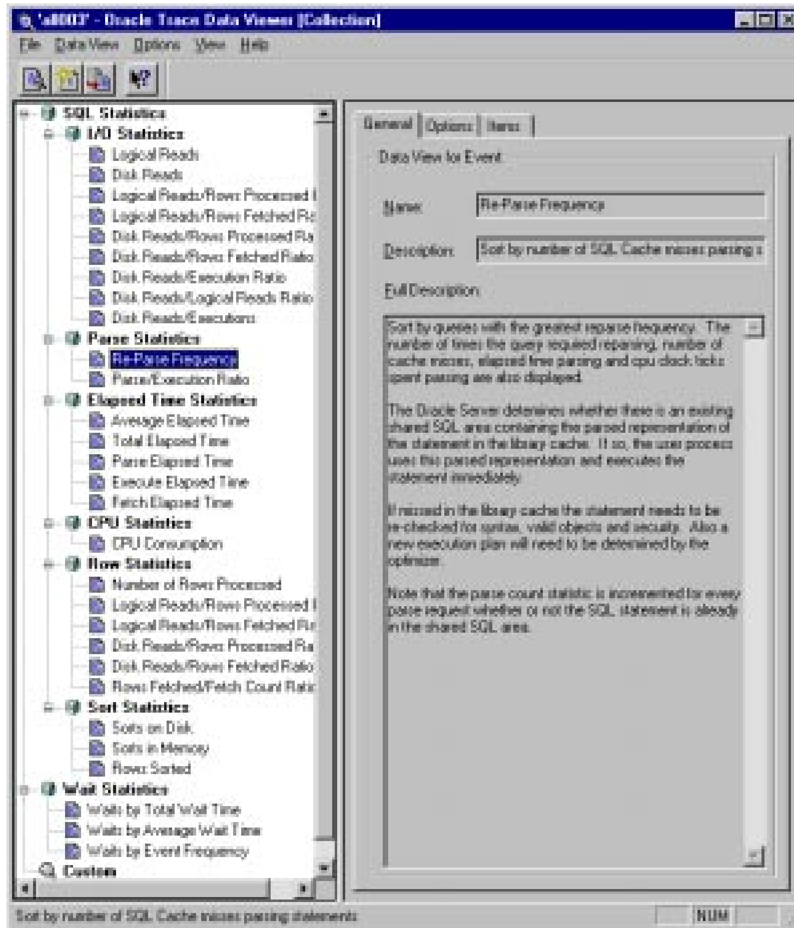


Table 14–1 explains the predefined data views shown in the previous figure as provided by Oracle Trace.

Table 14–1 Predefined Data Views Provided By Oracle Trace **(Page 1 of 5)**

View Name	Sort By	Data Displayed	Description
Logical Reads	Total number of logical reads performed for each distinct query.	Total number of blocks read during parses, executions and fetches. Logical reads for parses, executions and fetches of the query.	Logical data block reads include data block reads from both memory and disk. Input/output is one of the most expensive operations in a database system. I/O intensive statements can monopolize memory and disk usage causing other database applications to compete for these resources.
Disk Reads	Queries that incur the greatest number of disk reads.	Disk reads for parses, executions, and fetches.	Disk reads also known as physical I/O are database blocks read from disk. The disk read statistic is incremented once per block read regardless of whether the read request was for a multiblock read or a single block read. Most physical reads load data, index, and rollback blocks from the disk into the buffer cache. A physical read count can indicate a high miss rate within the data buffer cache.
Logical Reads/Rows Fetched Ratio	Number of logical reads divided by the number of rows fetched for all executions of the current query.	Total logical I/O. Total number of rows fetched.	The more blocks accessed relative to the number of rows actually returned the more expensive each row is to return. Can be a rough indication of relative expense of a query.
Disk Reads/Rows Fetched Ratio	Number of disk reads divided by the number of rows fetched for all executions of the current query.	Total disk I/O. Total number of rows fetched.	The greater the number of blocks read from disk for each row returned the more expensive each row is to return. Can be a rough indication of relative expense of a query.
Disk Reads/Execution Ratio	Total number of disk reads per distinct query divided by the number of executions of that query.	Total disk I/O. Logical I/O for the query as well as the number of executions of the query.	Indicates which statements incur the greatest number of disk reads per execution.

Table 14–1 Predefined Data Views Provided By Oracle Trace **(Page 2 of 5)**

View Name	Sort By	Data Displayed	Description
Disk Reads/Logical Reads Ratio	Greatest miss rate ratio of disk to logical reads.	Individual logical reads. Disk reads for the query as well as the miss rate.	The miss rate indicates the percentage of times the Oracle server needed to retrieve a database block on disk as opposed to locating it in the data buffer cache in memory. The miss rate for the data block buffer cache is derived by dividing the physical reads by the number of accesses made to the block buffer to retrieve data in consistent mode plus the number of blocks accessed through single block gets. Memory access is much faster than disk access, the greater the hit ratio, the better the performance.
Re-Parse Frequency	Queries with the greatest reparse frequency.	Number of cache misses. Total number of parses. Total elapsed time parsing. Total CPU clock ticks spent parsing.	The Oracle server determines whether there is an existing shared SQL area containing the parsed representation of the statement in the library cache. If so, then the user process uses this parsed representation and executes the statement immediately. If missed in the library cache, then re-check the statement for syntax, valid objects, and security. Also, the optimizer must determine a new execution plan. The parse count statistic is incremented for every parse request, regardless of whether the SQL statement is already in the shared SQL area.
Parse/Execution Ratio	Number of parses divided by the number executions per statement.	Individual number of parses. Number of executions.	The count of parses to executions should be as close to one as possible. If there are a high number of parses per execution, then the statement has been needlessly reparsed. This could indicate the lack of use of bind variables in SQL statements or poor cursor reuse. Reparsing a query means that the SQL statement has to be re-checked for syntax, valid objects and security. Also, a new execution plan needs to be determined by the optimizer.

Table 14–1 Predefined Data Views Provided By Oracle Trace (Page 3 of 5)

View Name	Sort By	Data Displayed	Description
Average Elapsed Time	Greatest average time spent parsing, executing and fetching on behalf of the query.	Individual averages for parse, execution and fetch.	The average elapsed time for all parses, executions and fetches-per-execution are computed, then summed for each distinct SQL statement in the collection.
Total Elapsed Time	Greatest total elapsed time spent parsing, executing and fetching on behalf of the query.	Individual elapsed times for parses, executions and fetches.	The total elapsed time for all parses, executions and fetches are computed, then summed for each distinct SQL statement in the collection.
Parse Elapsed Time	Total elapsed time for all parses associated with a distinct SQL statement.	SQL cache misses. Elapsed times for execution and fetching. Total elapsed time.	During parsing the Oracle server determines whether there is an existing shared SQL area containing the parsed representation of the statement in the library cache. If so, then the user process uses this parsed representation and executes the statement immediately. If missed in the library cache, then the statement needs to be rechecked for syntax, valid objects and security. Also, a new execution plan needs to be determined by the optimizer.
Execute Elapsed Time	Greatest total elapsed time for all executions associated with a distinct SQL statement.	Total elapsed time. Individual elapsed times for parsing and fetching.	The total elapsed time of all execute events for all occurrences of the query within an Oracle Trace collection.
Fetch Elapsed Time	Greatest total elapsed time for all fetches associated with a distinct SQL statement.	Number of rows fetched. Number of fetches. Number of executions. Total elapsed time. Individual elapsed times for parsing and executing.	The total elapsed time spent fetching data on behalf of all occurrences of the current query within the Oracle Trace collection.

Table 14–1 Predefined Data Views Provided By Oracle Trace **(Page 4 of 5)**

View Name	Sort By	Data Displayed	Description
CPU Statistics	Total CPU clock ticks spent parsing, executing and fetching on behalf of the SQL statement.	CPU clock ticks for parses, executions and fetches. Number of SQL cache misses and sorts in memory.	When SQL statements and other types of calls are made to an Oracle server, a certain amount of CPU time is necessary to process the call. Average calls require a small amount of CPU time. However, a SQL statement involving a large amount of data, a runaway query, in memory sorts or excessive reparsing can potentially consume a large amount of CPU time. CPU time displayed is in terms of the number of CPU clock ticks on the operating system housing the database.
Number of Rows Returned	Greatest total number of rows returned during execution and fetch for the SQL statement.	Number of rows returned during the fetch operation as well as the execution rows.	Targets queries that manipulate the greatest number of rows during fetching and execution. May mean that high gains can be made by tuning row intensive queries.
Rows Fetched/Fetch Count Ratio	Number of rows fetched divided by the number of fetches.	Individual number of rows fetched. Number of fetches.	This ratio shows how many rows were fetched at a time. It may indicate the level to which array fetch capabilities have been utilized. A ratio close to one may indicate an opportunity to optimize code by using array fetches.
Sorts on Disk	Queries that did the greatest number of sorts on disk.	Sort statistics for SQL statements. Number of in memory sorts. Total number of rows sorted.	Sorts on disk are sorts that could not be performed in memory, therefore they are more expensive because memory access is faster than disk access.
Sorts in Memory	Queries that did the greatest number of sorts in memory.	Sort statistics for SQL statements. Number of disk sorts. Total number of rows sorted.	Sorts in memory are sorts that could be performed completely within the sort buffer in memory without using the temporary tablespace segments.
Rows Sorted	Queries that sorted the greatest number of rows.	Number of in memory sorts. Number of sorts on disk.	Returns sort statistics for SQL statements ordered by queries that sorted the greatest number of rows.

Table 14–1 Predefined Data Views Provided By Oracle Trace **(Page 5 of 5)**

View Name	Sort By	Data Displayed	Description
Waits by Total Wait Time	Highest total wait time per distinct type of wait.	Average wait time, total wait time and number of waits per wait type.	Waits are sorted by wait description or type that had the greatest cumulative wait time for all occurrences of the wait type within the collection.
Waits by Average Wait Time	Highest average wait time per wait type.	Average wait time, total wait time and number of waits per wait type.	Waits are sorted by wait description or type that had the greatest average wait time for all occurrences of the wait type within the collection.
Waits by Event Frequency	Frequency of waits per wait type.	Number of waits per wait type, average wait time, and total wait time.	Waits are sorted by wait events or wait descriptions that appear most frequently within the collection.

Viewing Oracle Trace Data

Double clicking on SQL or Wait event data views provided by Data Viewer causes Oracle Trace to query the collection data and display data sorted by criteria described in the data view's description.

For example, double clicking the "Disk Reads/Log Reads Ratio" view returns data sorted by queries with the highest data buffer cache miss rate. This also displays the individual disk and logical read values.

Double clicking the "Average Elapsed Time" data view returns data sorted by queries that took the greatest average elapsed time to parse, execute, and fetch. It also displays the average elapsed times for parsing, execution, and fetching.

Figure 14–3 shows data in the "Average Elapsed Time" data view. Query text and statistics appear in the top portion of the window. Clicking any column headers causes the Data Viewer to sort rows by the statistic in that column.

Figure 14–3 Oracle Trace Data Viewer - Data View Screen

The currently selected data view's SQL text is shown in the lower portion of the window in the SQL Statement property sheet. Full statistical details about the currently selected data view also appear in the Details property sheet.

When examining a data view like that shown in Figure 14–3, you can print either of the following:

- Data view statistics, located in the top portion of the screen.
- Current SQL statement text in formatted output plus details on all statistical data collected for the currently selected query, located in the Details property page.

Window focus at the time of printing determines which portion of the screen is printed. For example, if focus is on the top portion of the screen, then the tabular form of all statistics and SQL for this data view is printed.

SQL Statement Property Page

The SQL Statement property page displays the currently selected query in a formatted output.

Details Property Page

The Details property page displays a detailed report on statistics for all executions of a given query within an Oracle Trace collection. Text for the currently selected SQL statement is posted at the end of the property page.

Example of Details Property Page

Statistics for all parses, executions, and fetches of the SQL statement.

The number of misses in library cache during Parse: 1.000000

Elapsed time statistics for the SQL statement:

```
Average Elapsed Time:     0.843000
Total Elapsed Time:       0.843000

Total Elapsed Parse:      0.000000
Total Elapsed Execute:    0.843000
Total Elapsed Fetch:      0.000000

Average Elapsed Parse:    0.000000
Average Elapsed Execute:  0.843000
Average Elapsed Fetch:    0.000000
```

Number of times parse, execute and fetch were called:

```
Number of Parses:       1
Number of Executions:   1
Number of Fetches:      0
```

Logical I/O statistics for parse, execute and fetch calls:

```
Logical I/O for Parses:              1
Logical I/O for Executions:        247
Logical I/O for Fetches:             0
Logical I/O Total:                 248
```

Disk I/O statistics for parse, execute and fetch calls:

```
Disk I/O for Parses:              0
Disk I/O for Executions:         28
Disk I/O for Fetches:             0
Disk I/O Total:                  28
```

CPU statistics for parse, execute and fetch calls:

```
CPU for Parses:              0
CPU for Executions:      62500
CPU for Fetches:             0
CPU Total:               62500
```

Row statistics for execute and fetch calls:

```
Rows processed during Executions:        104
Rows processed during Fetches:             0
Rows Total:                              104
```

Sort statistics for execute and fetch calls:

```
Sorts on disk:             0
Sorts in memory:           2
Sort rows:               667
```

Hit Rate - Disk I/O divided by Logical I/O: 0.112903

Logical I/O performed divided by rows actually processed: 2.384615

Disk I/O performed divided by number of executions: 28.000000

The number of parses divided by number of executions: 1.000000

The number of rows fetched divided by the number of fetches: 0.000000

```
   INSERT INTO tdv_sql_detail
    (collection_number, sql_text_hash,
     "LIB_CACHE_ADDR")
    SELECT DISTINCT collection_number,
                    sql_text_hash,
                    "LIB_CACHE_ADDR"
        FROM v_192216243_f_5_e_7_8_0
        WHERE collection_number = :b1;
```

Getting More Information on a Selected Query

There are two convenient ways to obtain additional data for the currently selected SQL statement:

- To modify a data view to add or remove statistics or items, select Modify from the Data View menu. You may add or remove statistics in the Items property sheet. These statistics appear as new columns in the data view. The selected query in Figure 14–3 is:

  ```
  SELECT COUNT(DISTINCT WAIT_TIME)
  FROM WAITS
  WHERE COLLECTION_NUMBER = :1;
  ```

  This query counts distinct values in the `wait_time` column of the `waits` table. By modifying the existing data view you can add other statistics that may be of interest such as "Execute Rows", which is the number of rows processed during execution, or "Execute CPU", which is the number of CPU clock ticks during execution.

  You can also remove existing columns, change the sort order, or change the default number of rows to view. You can save the modified data view. Oracle stores user-defined data views in the Custom data view container following the Data Viewer supplied list of SQL and Wait data views.

- Drill to statistics on all parses, executions and fetches of the selected query by clicking the Drill icon in the toolbar. The Drill down Data View dialog is displayed as shown in Figure 14–4.

Figure 14–4 Oracle Trace Data Viewer - Drill Down Data View Screen

Drill-down data views show individual statistics for all parses, executions, and fetches.

In Figure 14–4 the "Basic Statistics for Parse/Execute/Fetch" drill-down data view is selected. It displays statistics similar to those from TKPROF.

Note: For more information on TKPROF, see Chapter 11, "Overview of Diagnostic Tools".

Table 14–2 Drill-down Data Views **(Page 1 of 2)**

Drill-down Name	Sort By	Data Displayed	Description
Basic Statistics for Parse/Execute/Fetch	Greatest elapsed time	For each distinct call:. CPUs Elapsed time Disk I/O Logical I/O Number of rows processed	Parse, execution, and fetch statistics which are similar to statistics from `TKPROF`.
CPU Statistics for Parse/Execute/Fetch	Greatest number of CPUs	CPU total Pagefaults	CPU and pagefault statistics for parses, executions, and fetches of the current query. CPU total is the number of clock ticks in both user and system mode. The clock tick granularity is specific to the operating system on which the database resides.
I/O Statistics for Parse/Execute/Fetch	Greatest number of disk I/Os	Logical and Disk I/O statistics Pagefault I/O (number of hard pagefaults) Input I/O (number of times the file system performed input) Output I/O (number of times the file system performed output)	I/O statistics for parses, executions, and fetches.
Parse Statistics	Greatest elapsed time	Current user identifier Schema identifiers	Parse statistics, for example, whether the current statement was missed in library cache, Oracle optimizer mode, current user identifier, and schema identifier.

Table 14–2 Drill-down Data Views **(Page 2 of 2)**

Drill-down Name	Sort By	Data Displayed	Description
Row Statistics for Execute/Fetch	Greatest number of rows returned	Number of rows returned Number of rows sorted Number of rows returned during a full table scan	Execution and fetch row statistics.
Sort Statistics for Parse/Execute/Fetch	Greatest elapsed time	Sorts on disk Sorts in memory Number of rows sorted Number of rows returned from a full table scan	Parse, execution, and fetch sort statistics.
Wait Parameters	Wait_time	Description Wait_time P1 P2 P3	Investigating waits may help identify sources of contention. P1, P2, and P3 parameters are values that provide more information about specific wait events. The parameters are foreign keys to views that are wait event dependent. For example, for latch waits, P2 is the latch number that is a foreign key to `V$LATCH`. The meaning of each parameter is specific to each wait type.

Manually Collecting Oracle Trace Data

Though the Oracle Trace Manager is the primary interface to Oracle Trace, you can optionally force a manual collection of Oracle Trace data. You can do this by using a command-line interface, editing initialization parameters, or by executing stored procedures.

Using the Oracle Trace Command-Line Interface

Another option for controlling Oracle Trace server collections is the Oracle Trace CLI (Command-line Interface). The CLI collects event data for all server sessions attached to the database at collection start time. Sessions that attach after the collection is started are excluded from the collection. The CLI is invoked by the `OTRCCOL` statement for the following functions:

- OTRCCOL START *job_id input_parameter_file*
- OTRCCOL STOP *job_id input_parameter_file*
- OTRCCOL FORMAT *input_parameter_file*
- OTRCCOL DCF col_name cdf_file
- OTRCCOL DFD col_name username password service

The parameter job_id can be any numeric value, but it must be unique and you must remember this value to stop the collection. The input parameter file contains specific parameter values required for each function as shown in the following examples. col_name (collection name) and cdf_file (collection definition file) are initially defined in the START function input parameter file.

The OTRCCOL START statement invokes a collection based upon parameter values contained in the input parameter file. For example:

```
OTRCCOL START 1234 my_start_input_file
```

Where file my_start_input_file contains the following input parameters:

col_name	*my_collection*
dat_file	<usually same as collection name>.dat
cdf_file	<usually same as collection name>.cdf
fdf_file	<server event set>.fdf
regid	1 192216243 0 0 5 <database SID>

The server event sets that can be used as values for the fdf_file are ORACLE, ORACLEC, ORACLED, ORACLEE, and ORACLESM.

> **See Also:** For more information on the server event sets, see "Using Initialization Parameters to Control Oracle Trace" on page 14-22.

The OTRCCOL STOP statement halts a running collection as follows:

```
OTRCCOL STOP 1234 my_stop_input_file
```

Where my_stop_input_file contains the collection name and cdf_file name.

The OTRCCOL FORMAT statement formats the binary collection file to Oracle tables. An example of the FORMAT statement is:

```
OTRCCOL FORMAT my_format_input_file
```

Where `my_format_input_file` contains the following input parameters:

username	<database username>
password	<database password>
service	<database service name>
cdf_file	<usually same as collection name>.cdf
full_format	<0/1>

A `full_format` value of 1 produces a full format; a value of 0 produces a partial format.

> **See Also:** For more information on formatting part or all of an Oracle Trace collection and for other important information on creating the Oracle Trace formatting tables prior to running the format statement, see "Formatting Oracle Trace Data to Oracle Tables" on page 14-27.

The `OTRCCOL DCF` statement deletes collection files for a specific collection. The `OTRCCOL DFD` statement deletes formatted data from the Oracle Trace formatter tables for a specific collection.

Using Initialization Parameters to Control Oracle Trace

Six parameters are set up by default to control Oracle Trace. By logging into the administrator account in your database and executing the `SHOW PARAMETERS TRACE` statement, you see the following parameters as shown in Table 14–3:

Table 14–3 Oracle Trace Initialization Parameters

Name	Type	Value
ORACLE_TRACE_COLLECTION_NAME	string	[null]
ORACLE_TRACE_COLLECTION_PATH	string	$ORACLE_HOME/otrace/admin/cdf
ORACLE_TRACE_COLLECTION_SIZE	integer	5242880
ORACLE_TRACE_ENABLE	boolean	FALSE
ORACLE_TRACE_FACILITY_NAME	string	oracled
ORACLE_TRACE_FACILITY_PATH	string	$ORACLE_HOME/otrace/admin/cdf

You can modify the Oracle Trace initialization parameters and use them by adding them to your initialization file.

> **See Also:** This chapter references file path names on UNIX-based systems. For the exact path on other operating systems, see your Oracle platform-specific documentation. A complete discussion of these parameters is provided in *Oracle8i Reference.*

Enabling Oracle Trace Collections

The `ORACLE_TRACE_ENABLE` parameter is set to `false` by default. A value of `FALSE` disables any use of Oracle Trace for that Oracle server.

To enable Oracle Trace collections for the server, set the parameter to `true`. Having the parameter set to `true` does not start an Oracle Trace collection, but instead allows Oracle Trace to be used for that server. You can then start Oracle Trace in one of the following ways:

- Using the Oracle Trace Manager application supplied with the Oracle Diagnostics Pack.
- Setting the `ORACLE_TRACE_COLLECTION_NAME` parameter.

When `ORACLE_TRACE_ENABLE` is set to `true`, you can start and stop an Oracle Trace server collection by either using the Oracle Trace Manager application that is supplied with the Oracle Diagnostics Pack, or you can enter a collection name in the `ORACLE_TRACE_COLLECTION_NAME` parameter. The default value for this parameter is `null`. A collection name can be up to 16 characters in length. You must then shut down your database and start it up again to activate the parameters. If a collection name is specified, then when you start the server, you automatically start an Oracle Trace collection for all database sessions, which is similar in functionality to SQL Trace.

To stop the collection that was started using the `ORACLE_TRACE_COLLECTION_NAME` parameter, shut down the server instance and reset the `ORACLE_TRACE_COLLECTION_NAME` to `null`. The collection name specified in this value is also used in two collection output file names: the collection definition file (*collection_name*.`cdf`) and the binary data file (*collection_name*.`dat`).

Determining the Event Set that Oracle Trace Collects

The `ORACLE_TRACE_FACILITY_NAME` initialization parameter specifies the event set that Oracle Trace collects. The name of the `DEFAULT` event set is `ORACLED`. The `ALL` event set is `ORACLE`, the `EXPERT` event set is `ORACLEE`, the `SUMMARY` event set is `ORACLESM`, and the `CACHEIO` event set is `ORACLEC`.

After it is restarted, if the database does not begin collecting data, then check the following:

- The event set file, identified by ORACLE_TRACE_FACILITY_NAME, with .fdf appended to it, should be in the directory specified by the ORACLE_TRACE_FACILITY_PATH initialization parameter. The exact directory that this parameter specifies is platform-specific.
- The following files should exist in your Oracle Trace administrative directory: REGID.DAT, PROCESS.DAT, and COLLECT.DAT. If they do not, then you must run the OTRCCREF executable to create them.
- The Oracle Trace parameters should be set to the values that you changed in the initialization file. Use Instance Manager to identify Oracle Trace parameter settings.
- Look for an EPC_ERROR.LOG file to see more information about why a collection failed. Oracle Trace creates the EPC_ERROR.LOG file in the current default directory of the Oracle Intelligent Agent when it runs the Oracle Trace Collection Services OTRCCOL image. Depending on whether you are running Oracle Trace from the Oracle Trace Manager or from the command-line interface, you can find the EPC_ERROR.LOG file in one of the following locations:
  - $ORACLE_HOME or $ORACLE_HOME/network/agent on UNIX
  - %ORACLE_HOME%\network\agent or %ORACLE_HOME%\net80\agent on NT
  - $ORACLE_HOME\rdbmsnn on NT or $ORACLE_HOME\rdbms on UNIX
  - In your current working directory, if you are using the command-line interface
  - To find the EPC_ERROR.LOG file on UNIX, change directories to the $ORACLE_HOME directory and execute the statement:

    ```
    find . -name EPC_ERROR.LOG -print .
    ```

    Note: On UNIX, the EPC_ERROR.LOG file name is case sensitive and is in uppercase.

  - Look for *.trc files in the directory specified by the USER_DUMP_DEST initialization parameter. Searching for "epc" in the *.trc files may give

errors. These errors and their descriptions are located in the $ORACLE_HOME/otrace/include/epc.h file.

Using Stored Procedures to Control Oracle Trace

Using the Oracle Trace stored procedures you can invoke an Oracle Trace collection for your own session or for another session. To collect Oracle Trace data for your own database session, execute the following stored procedure package syntax:

```
DBMS_ORACLE_TRACE_USER.SET_ORACLE_TRACE(true/false,
COLLECTION_NAME, SERVER_EVENT_SET)
```

where:

true/false	Boolean: true to activate, false to deactivate.
COLLECTION_NAME	VARCHAR2: collection name (no file extension, eight character maximum).
SERVER_EVENT_SET	VARCHAR2: server event set file name (CONNECT, ORACLE, ORACLEC, ORACLED, ORACLEE, ORACLSM, SQL_ONLY, SQL_PLAN, SQL_TXN, SQL_WAITS, or WAITS). See Table 14–4 for a description of each of these event set file names.

Example:

```
EXECUTE DBMS_ORACLE_TRACE_USER.SET_ORACLE_TRACE (true,'MYCOLL','oracle');
```

Table 14–4 Server Event Set File Names

Event Set File Name (.fdf)	Description
CONNECT	CONNECT_DISCONNECT event set. Collects statistics about connects to the database and disconnects from the database.
ORACLE	ALL event set. Collects all statistics for the Oracle Server including wait events.
ORACLEC	CACHEIO event set. Collects caching statistics for buffer cache I/O.
ORACLED	Oracle Server DEFAULT event set. Collects statistics for the Oracle Server.
ORACLEE	EXPERT event set. Collects statistics for the Oracle Expert application.

Table 14–4 Server Event Set File Names (Cont.)

Event Set File Name (.fdf)	Description
ORACLESM	SUMMARY event set. Collects workload statistics for the Summary Advisor application.
SQL_ONLY	SQL_TEXT_ONLY event set. Collects statistics about connects to the database, disconnects from the database, and SQL text.
SQL_PLAN	SQL_STATS_AND_PLAN event set. Collect statistics about connects to the database, disconnects from the database, SQL statistics, SQL text, and row source (EXPLAIN PLAN).
SQL_TXN	SQL_TXNS_AND_STATS event set. Collects statistics about connects to the database, disconnects from the database, transactions, SQL text and statistics, and row source (EXPLAIN PLAN).
SQL_WAITS	SQL_AND_WAIT_STATS event set. Collects statistics about connects to the database, disconnects from the database, row source (EXPLAIN PLAN), SQL text and statistics, and wait events.
WAITS	WAIT_EVENTS event set. Collects statistics about connects to the database, disconnects from the database, and wait events.

To collect Oracle Trace data for a database session other than your own, execute the following stored procedure package syntax:

```
DBMS_ORACLE_TRACE_AGENT.SET_ORACLE_TRACE_IN_SESSION
(sid, serial#, true/false, COLLECTION_NAME, SERVER_EVENT_SET)
```

Where:

sid	Number: session instance from V$SESSION.SID.
serial#	Number: session serial number from V$SESSION.SERIAL#.

Example:

```
EXECUTE DBMS_ORACLE_TRACE_AGENT.SET_ORACLE_TRACE_IN_SESSION
(8,12,true,'NEWCOLL','oracled');
```

If the collection does not occur, then check the following:

- Be sure the server event set file identified by SERVER_EVENT_SET exists. If there is no full file specification on this field, then the file should be located in

the directory identified by ORACLE_TRACE_FACILITY_PATH in the database initialization file.

- The following files should exist in your Oracle Trace admin directory: REGID.DAT, PROCESS.DAT, and COLLECT.DAT. If they do not, then you must run the OTRCCREF executable to create them.
- For Oracle Server release 8.0 and later, the stored procedure packages do not exist in the database. If the packages do not exist, then run the OTRCSVR.SQL file (in your Oracle Trace admin directory) to create the packages.
- The user has the EXECUTE privilege on the stored procedure.

Oracle Trace Collection Results

Running an Oracle Trace collection produces the following collection files:

- COLLECTION_NAME.CDF is the Oracle Trace collection definition file for your collection.
- COLLECTION_NAME.DAT files are the Oracle Trace output files containing the trace data in binary format.

You can access the Oracle Trace data in the collection files in the following ways:

- You can create Oracle Trace reports from the binary file.
- The data can be formatted to Oracle tables for Data Viewer, SQL access, and reporting.

Formatting Oracle Trace Data to Oracle Tables

You can format Oracle Trace server collection Oracle tables for more flexible access SQL reporting tools. Oracle Trace produces a separate table for each event collected. For example, a parse event table is created to store data for all parse events occurring during a server collection. Before you can format data, you must first set up the Oracle Trace formatter tables by executing the OTRCFMTC.SQL script on the server host machine.

Note: Oracle server releases 7.3.4 and later automatically create the formatter tables.

Use the following syntax to format an Oracle Trace collection:

```
OTRCFMT [optional parameters] collection_name.cdf [user/password@database]
```

If you omit `user/password@database`, then Oracle prompts you for this information.

Oracle Trace allows data to be formatted while a collection is occurring. By default, Oracle Trace formats only the portion of the collection that has not been formatted previously. If you want to reformat the entire collection file, then use the optional parameter `-f`.

Oracle Trace provides several SQL scripts that you can use to access the server event tables. For more information on server event tables and scripts for accessing event data and improving event table performance, refer to the *Oracle Trace User's Guide*.

Oracle Trace Statistics Reporting Utility

The Oracle Trace statistics reporting utility displays statistics for all items associated with each occurrence of a server event. These reports can be quite large. You can control the report output by using statement parameters. Use the following statement and optional parameters to produce a report:

```
OTRCREP  [optional parameters]  collection_name.CDF
```

First, you may want to run a report called "`PROCESS.txt`". You can produce this report to provide a listing of specific process identifiers for which you want to run another report.

You can manipulate the output of the Oracle Trace reporting utility by using the following optional report parameters:

`output_path` Specifies a full output path for the report files. If not specified, then the files are placed in the current directory.

`-p` Organizes event data by process. If you specify a process ID (pid), then you have one file with all the events generated by that process in chronological order. If you omit the process ID, then you have one file for each process that participated in the collection. The output files are named *`collection_Ppid`*`.txt`.

`-P`	Produces a report called *`collection`*`_PROCESS.txt` that lists all processes that participated in the collection. It does not include event data. You could produce this report first to determine the specific processes for which you might want to produce more detailed reports.
`-w#`	Sets report width, such as -w132. The default is 80 characters.
`-l#`	Sets the number of report lines per page. The default is 63 lines per page.
`-h`	Suppresses all event and item report headers, producing a shorter report.
`-s`	Used with Net8 data only. This option creates a file similar to the SQL*Net Tracing file.
`-a`	Creates a report containing all the events for all products, in the order they occur in the data collection (`.dat`) file.

15

Dynamic Performance Views

Dynamic performance views, or "V$" views, are useful for identifying instance-level performance problems. All V$ views are listed in the V$FIXED_TABLE view.

V$ view content is provided by underlying X$ tables. The X$ tables are internal data structures that can be modified by SQL statements. These tables are therefore only available when an instance is in a NOMOUNT or MOUNT state.

This chapter describes the most useful V$ views for performance tuning. V$ views are also useful for ad hoc investigation, for example, when users report sudden response time deterioration.

Although the V$ views belong to user SYS, users other than SYS have read-only access to V$ views. Oracle populates the V$ views and X$ tables at instance startup. Their contents are flushed when you shut down the instance.

The X$ tables and their associated V$ views are dynamic, so their contents are constantly changing. X$ tables retain timing information providing you have set the initialization parameter TIMED_STATISTICS to true, or if you execute the SQL statement:

```
ALTER SYSTEM SET TIMED_STATISTICS=true;
```

This chapter contains the following sections:

- Instance-Level Views for Tuning
- Session-Level or Transient Views for Tuning
- Current Statistic Values and Rates of Change

See Also: For complete information on all dynamic performance tables, please see the *Oracle8i Reference.*

Instance-Level Views for Tuning

These views concern the instance as a whole and record statistics either since startup of the instance or (in the case of the SGA statistics) the current values, which remains constant until altered by some need to reallocate SGA space. Cumulative statistics are from startup.

Table 15–1 Instance Level Views Important for Tuning

View	Notes
V$FIXED_TABLE	Lists the fixed objects present in the release.
V$INSTANCE	Shows the state of the current instance.
V$LATCH	Lists statistics for nonparent latches and summary statistics for parent latches.
V$LIBRARYCACHE	Contains statistics about library cache performance and activity.
V$ROLLSTAT	Lists the names of all online rollback segments.
V$ROWCACHE	Shows statistics for data dictionary activity.
V$SGA	Contains summary information on the system global area.
V$SGASTAT	Contains detailed information on the system global area.
V$SORT_USAGE	Shows the size of the temporary segments and the session creating them. This information can help you identify which processes are doing disk sorts.
V$SQLAREA	Lists statistics on shared SQL area; contains one row per SQL string. Provides statistics on SQL statements that are in memory, parsed, and ready for execution. Text limited to 1000 characters; full text is available in 64 byte chunks from V$SQLTEXT.
V$SQLTEXT	Contains the text of SQL statements belonging to shared SQL cursors in the SGA.
V$SYSSTAT	Contains basic instance statistics.
V$SYSTEM_EVENT	Contains information on total waits for an event.
V$WAITSTAT	Lists block contention statistics. Updated only when timed statistics are enabled.

The single most important fixed view is V$SYSSTAT, which contains the statistic name in addition to the value. The values from this table form the basic input to the instance tuning process.

Session-Level or Transient Views for Tuning

These views either operate at the session level or primarily concern transient values. Session data is cumulative from connect time.

Table 15–2 Session Level Views Important for Tuning

View	Notes
V$LOCK	Lists the locks currently held by the Oracle8 Server and outstanding requests for a lock or latch.
V$MYSTAT	Shows statistics from your current session.
V$PROCESS	Contains information about the currently active processes.
V$SESSION	Lists session information for each current session. Links SID to other session attributes. Contains row lock information.
V$SESSION_EVENT	Lists information on waits for an event by a session.
V$SESSION_WAIT	Lists the resources or events for which active sessions are waiting, where WAIT_TIME = 0 for current events.
V$SESSTAT	Lists user session statistics. Requires join to V$STATNAME, V$SESSION.

The structure of V$SESSION_WAIT makes it easy to check in real time whether any sessions are waiting, and if so, why. For example:

```
SELECT SID,EVENT
    FROM V$SESSION_WAIT
    WHERE WAIT_TIME = 0;
```

You can then investigate to see whether such waits occur frequently and whether they can be correlated with other events, such as the use of particular modules.

Current Statistic Values and Rates of Change

This section describes procedures for:

- Finding the Current Value of a Statistic
- Finding the Rate of Change of a Statistic

Finding the Current Value of a Statistic

Key ratios are expressed in terms of instance statistics. For example, the consistent change ratio is consistent changes divided by consistent gets. The simplest effective SQL*Plus script for finding the current value of a statistic is of the form:

```
COL name FORMAT a35
COL value FORMAT 999,999,990
SELECT name, value
FROM V$SYSSTAT S
WHERE lower(NAME) LIKE lower('%&stat_name%')
/
```

> **Note:** Two `LOWER` functions in the preceding query make it case insensitive and allow it to report data from the 11 statistics whose names start with "CPU" or "DBWR". No other upper-case characters appear in statistic names.

You can use the following query, for example, to report all statistics containing the word "get" in their name:

```
@STAT GET
```

It is preferable, however, to use mechanisms that record the change in the statistic(s) over a known period of time as described in the next section of this chapter.

Finding the Rate of Change of a Statistic

You can adapt the following script to show the rate of change for any statistic, latch, or event. For a given statistic, this script tells you the number of seconds between two checks of its value, and its rate of change.

```
SET VERI OFF
DEFINE secs=0
DEFINE value=0
COL value FORMAT 99,999,999,990 new_value value
COL secs FORMAT a10 new_value secs noprint
COL delta FORMAT 9,999,990
COL delta_time FORMAT 9,990
COL rate FORMAT 999,990.0
COL name FORMAT a30
SELECT name, value, TO_CHAR(sysdate,'sssss') secs,
    (value - &value) delta,
    (TO_CHAR(sysdate,'sssss') - &secs) delta_time,
    (value - &value)/ (TO_CHAR(sysdate,'sssss') - &secs) rate
    FROM v$sysstat
    WHERE name = '&&stat_name'
/
```

Note: Run this script at least twice, because the first time you run it, it initializes the SQL*Plus variables.

16

Diagnosing System Performance Problems

This chapter provides an overview of factors affecting performance in properly designed systems. Following the guidelines in this chapter cannot, however, compensate for poor design!

This chapter contains the following sections:

- Tuning Factors for Well Designed Existing Systems
- Insufficient CPU
- Insufficient Memory
- I/O Constraints
- Network Constraints
- Software Constraints

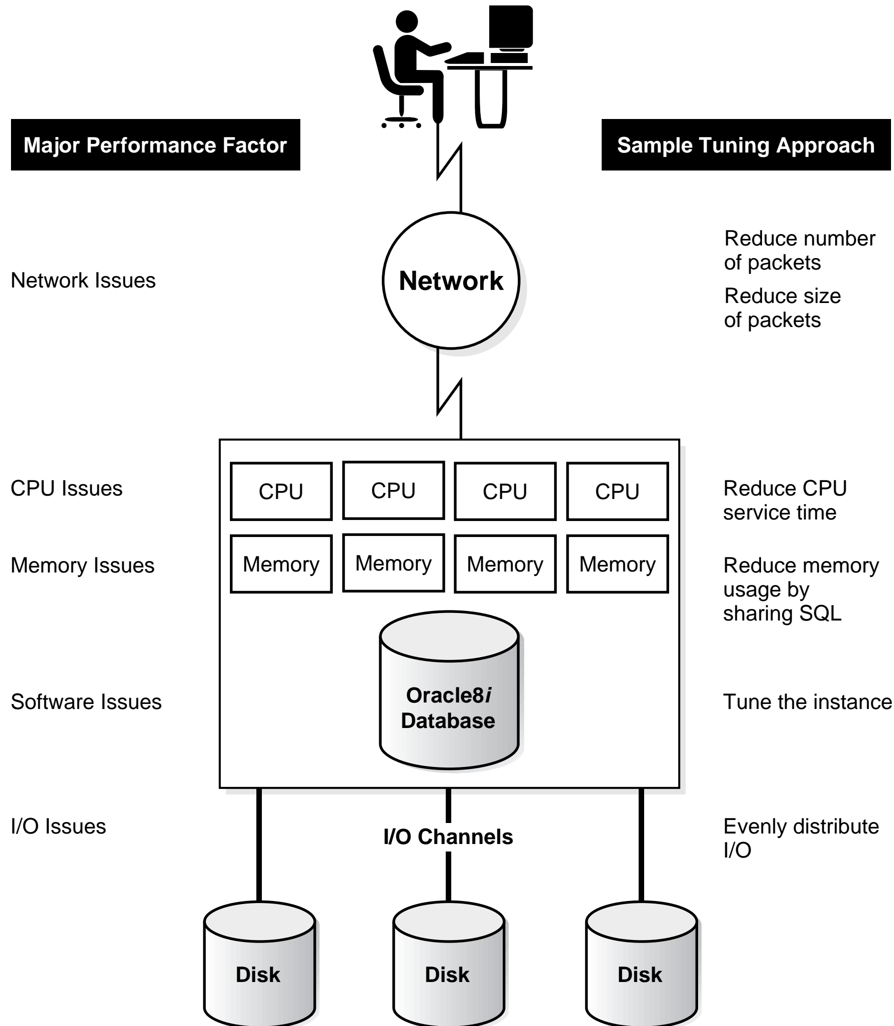
Note: Later chapters discuss each of these factors in depth.

Tuning Factors for Well Designed Existing Systems

Figure 16–1 illustrates the factors involved in Oracle system performance for well designed applications.

Note: Tuning these factors is effective only *after* you have tuned the business process and the application, as described in Chapter 2, "Performance Tuning Methods".

Figure 16–1 Major Performance Factors in Well Designed Systems



Performance problems tend to be interconnected rather than isolated and unrelated. Table 16–1 identifies the key performance factors in existing systems as well as the areas in which symptoms may appear. For example, buffer cache problems may show up as CPU, memory, or I/O problems. Therefore, tuning the buffer cache CPU may improve I/O.

Table 16–1 Key to Tuning Areas for Existing Systems

ORACLE TUNING AREAS	LIMITING RESOURCES				
	CPU	Memory	I/O	Network	Software
Application					
Design/Architecture	X	X	X	X	X
DML SQL	X	X	X	X	X
Query SQL	X	X	X	X	X
Client/server Roundtrips	X			X	
Instance					
Buffer Cache	X	X	X		
Shared Pool	X	X			
Sort Area	X	X	X		
Physical Structure of Data/DB File I/O	X		X		
Log File I/O		X	X		
Archiver I/O	X		X		
Rollback Segments			X		X
Locking	X	X	X		X
Backups	X		X	X	X
Operating System					
Memory Management	X	X	X		
I/O Management	X	X	X		
Process Management	X	X			
Network Management		X		X	

Insufficient CPU

In a CPU-bound system, CPU resources might be completely allocated, and service time could be excessive too. In this situation, you must improve your system's processing ability. Alternatively, you could have too much idle time, and the CPU might not be completely used up. In either case, you need to determine why so much time is spent waiting.

To determine why there is insufficient CPU, identify how your entire system is using CPU. Do not rely on identifying how CPU is used by Oracle server processes. At the beginning of a workday, for example, the mail system may consume a large amount of available CPU while employees check their messages. Later in the day, the mail system may be much less of a bottleneck, and its CPU use drops accordingly.

Workload is a very important factor when evaluating your system's level of CPU use. During peak workload hours, 90% CPU use with 10% idle and waiting time may be understandable and acceptable; 30% utilization at a time of low workload may also be understandable. However, if your system shows high utilization at normal workloads, then there is no more room for a "peak workload". You have a CPU problem if idle time and time waiting for I/O are both close to zero, or less than 5%, at a normal or low workload.

> **See Also:** For more information on CPU utilization, see Chapter 18, "Tuning CPU Resources".

Insufficient Memory

Sometimes a memory problem may be detected as an I/O problem. There are two types of memory requirements: Oracle and system. Oracle memory requirements affect the system requirements. Memory problems may be the cause of paging and swapping that occurs in the machine. So, make sure your system does not start paging and swapping. The system should be able to run within the limitations set by real memory.

System memory requirements for non-Oracle processes plus Oracle memory requirements should be equal to, or less than, the total available real memory. To achieve this, reduce the size of some of the Oracle memory structures, such as the buffer cache, shared pool, or the redo log buffer. On the system level, you can reduce the number of processes and/or the amount of memory each process uses. You can also identify which processes are using the most memory. One way to reduce memory use is by sharing SQL.

See Also: For more information on memory, see Chapter 19, "Tuning Memory Allocation".

I/O Constraints

Be sure to distribute I/O evenly across disks and channels. I/O constraints include:

- Channel bandwidth: the number of I/O channels.
- Device bandwidth: the number of disks.
- Device latency: the time elapsed from the initiation of a request to the receipt of the request; latency is part of the "wait time".

I/O problems may result from hardware limitations. Your system needs enough disks and SCSI busses to support the transaction throughput you need. You can evaluate the configuration by calculating the quantity of messages all your disks and busses can potentially support, and comparing that to the number of messages required by your peak workload.

If the response time of an I/O becomes excessive, the most common problem is that wait time has increased (response time = service time + wait time). If wait time increases, then there are too many I/O requests for this device. If service time increases, then the I/O requests are larger, so you write more bytes to disk.

The different background processes, such as DBWR, ARCH, and so on, perform different types of I/O, and each process has different I/O characteristics. Some processes read and write in the block size of the database, some read and write in larger chunks. If service time is too high, then stripe the file across different devices.

Mirroring can also be a cause of I/O bottlenecks, unless the data is mirrored to a destination database that has the same number of disks as the source database.

See Also: For more information on I/O bottlenecks, see Chapter 20, "Tuning I/O".

Network Constraints

Network constraints are similar to I/O constraints. You need to consider:

- Network bandwidth: Each transaction requires that a certain number of packets be sent over the network. If you know the number of packets required for one transaction, then you can compare that to the bandwidth to determine whether your system is capable of supporting the desired workload.

- Message rates: You can reduce the number of packets on the network by batching them, rather than sending many small packets.
- Transmission time.

As the number of users increases and demand rises, the network can quietly become the bottleneck in an application. You may spend a lot of time waiting for network availability. Use available operating system tools to see how busy your network is.

> **See Also:** For more information, see Chapter 22, "Tuning Networks".

Software Constraints

Operating system software determines:

- The maximum number of processes you can support.
- The maximum number of processes you can connect.

Before you can tune Oracle effectively, you should confirm that the operating system is performing optimally. Work closely with the hardware and software system administrators to ensure that Oracle is allocated the proper operating system resources.

> **Note:** On NT systems, there are no pre-set or configurable maximum numbers of processes that can be supported or connected.

> **See Also:** Operating system tuning is different for every platform. See your operating system documentation, as well as your Oracle operating system-specific documentation for more information. In addition, see Chapter 23, "Tuning the Operating System".

17

Transaction Modes

This chapter describes the different modes in which read consistency is performed.

This chapter contains the following sections:

- Using Discrete Transactions
- Using Serializable Transactions

Using Discrete Transactions

You can improve the performance of short, nondistributed transactions by using the `BEGIN_DISCRETE_TRANSACTION` procedure. This procedure streamlines transaction processing so that short transactions can execute faster.

This section describes:

- Deciding When to Use Discrete Transactions
- How Discrete Transactions Work
- Errors During Discrete Transactions
- Using Discrete Transactions
- Example

Deciding When to Use Discrete Transactions

Discrete transaction processing is useful for transactions that:

- Modify only a few database blocks.
- Never change an individual database block more than once per transaction.
- Do not modify data likely to be requested by long-running queries.
- Do not need to see the new value of data after modifying the data.
- Do not modify tables containing any `LONG` values.

In deciding to use discrete transactions, you should consider the following factors:

- Can the transaction be designed to work within the constraints placed on discrete transactions, as described in "Using Discrete Transactions" on page 17-3.
- Does using discrete transactions result in a significant performance improvement under normal usage conditions?

Discrete transactions can be used concurrently with standard transactions. Choosing whether to use discrete transactions should be a part of your normal tuning procedure. Discrete transactions can be used only for a subset of all transactions, for sophisticated users with advanced application requirements. However, where speed is the most critical factor, the performance improvements can justify the design constraints.

How Discrete Transactions Work

During a discrete transaction, all changes made to any data are deferred until the transaction commits. Redo information is generated, but it is stored in a separate location in memory.

When the transaction issues a commit request, the redo information is written to the redo log file (along with other group commits), and the changes to the database block are applied directly to the block. The block is written to the database file in the usual manner. Control is returned to the application after the commit completes. Oracle does not need to generate undo information, because the block is not actually modified until the transaction is committed, and the redo information is stored in the redo log buffers.

As with other transactions, the uncommitted changes of a discrete transaction are not visible to concurrent transactions. For regular transactions, undo information is used to re-create old versions of data for queries that require a consistent view of the data. Because no undo information is generated for discrete transactions, a discrete transaction that starts and completes during any query can cause the query to receive the "snapshot too old" error if the query requests data changed by the discrete transaction. For this reason, you might avoid performing queries that access a large subset of a table that is modified by frequent discrete transactions.

Errors During Discrete Transactions

Any errors encountered during processing of a discrete transaction cause the predefined exception `DISCRETE_TRANSACTION_FAILED` to be raised. These errors include the failure of a discrete transaction to comply with the usage notes outlined below. (For example, calling `BEGIN_DISCRETE_TRANSACTION` after a transaction has begun, or attempting to modify the same database block more than once during a transaction, raises the exception.)

Using Discrete Transactions

The `BEGIN_DISCRETE_TRANSACTION` procedure must be called before the first statement in a transaction. This call to the procedure is effective only for the duration of the transaction (that is, after the transaction is committed or rolled back, the next transaction is processed as a standard transaction).

Transactions that use this procedure cannot participate in distributed transactions.

Although discrete transactions cannot see their own changes, you can obtain the old value and lock the row, using the `FOR UPDATE` clause of the `SELECT` statement, before updating the value.

Because discrete transactions cannot see their own changes, a discrete transaction cannot perform inserts or updates on both tables involved in a referential integrity constraint.

For example, assume that the `emp` table has a `FOREIGN KEY` constraint on the `deptno` column that refers to the `dept` table. A discrete transaction cannot attempt to add a department into the `dept` table and then add an employee belonging to that department, because the department is not added to the table until the transaction commits, and the integrity constraint requires that the department exist before an insert into the `emp` table can occur. These two operations must be performed in separate discrete transactions.

Because discrete transactions can change each database block only once, some combinations of data manipulation statements on the same table are better suited for discrete transactions than others. One `INSERT` statement and one `UPDATE` statement used together are the least likely to affect the same block. Multiple `UPDATE` statements are also unlikely to affect the same block, depending on the size of the affected tables. Multiple `INSERT` statements (or `INSERT` statements that use queries to specify values), however, are likely to affect the same database block. Multiple DML operations performed on separate tables only affect the same database blocks if the tables are clustered.

Example

An application for checking out library books is an example of a transaction type that uses the `BEGIN_DISCRETE_TRANSACTION` procedure. The following procedure is called by the library application with the book number as the argument. This procedure checks to see if the book is reserved before allowing it to be checked out. If more copies of the book have been reserved than are available, then the status `RES` is returned to the library application, which calls another procedure to reserve the book, if desired. Otherwise, the book is checked out, and the inventory of books available is updated.

```
CREATE PROCEDURE checkout (bookno IN NUMBER (10)
                           status OUT VARCHAR(5))
AS
DECLARE
   tot_books   NUMBER(3);
   checked_out NUMBER(3);
   res         NUMBER(3);
BEGIN
   DBMS_TRANSACTION.BEGIN_DISCRETE_TRANSACTION;
   FOR i IN 1 .. 2 LOOP
      BEGIN
         SELECT total, num_out, num_res
           INTO tot_books, checked_out, res
           FROM books
           WHERE book_num = bookno
           FOR UPDATE;
      IF res >= (tot_books - checked_out)
      THEN
         status := 'RES';
      ELSE
         UPDATE books SET num_out = checked_out + 1
            WHERE book_num = bookno;
         status := 'AVAIL'
      ENDIF;
      COMMIT;
      EXIT;
   EXCEPTION
      WHEN DBMS_TRANSACTION.DISCRETE_TRANSACTION_FAILED THEN
         ROLLBACK;
      END;
   END LOOP;
END;
```

For the above loop construct, if the `DISCRETE_TRANSACTION_FAILED` exception occurs during the transaction, then the transaction is rolled back, and the loop executes the transaction again. The second iteration of the loop is not a discrete transaction, because the `ROLLBACK` statement ended the transaction; the next transaction processes as a standard transaction. This loop construct ensures that the same transaction is attempted again in the event of a discrete transaction failure.

Using Serializable Transactions

Oracle allows application developers to set the isolation level of transactions. The isolation level determines what changes the transaction and other transactions can see. The ISO/ANSI SQL3 specification details the following levels of transaction isolation.

SERIALIZABLE	Transactions lose no updates, provide repeatable reads, and do not experience phantoms. Changes made to a serializable transaction are visible only to the transaction itself.
READ COMMITTED	Transactions do not have repeatable reads, and changes made in this transaction or other transactions are visible to all transactions. This is the default transaction isolation.

If you want to set the transaction isolation level, then you must do so before the transaction begins. Use the `SET TRANSACTION ISOLATION LEVEL` statement for a particular transaction, or use the `ALTER SESSION SET ISOLATION_LEVEL` statement for all subsequent transactions in the session.

> **See Also:** *Oracle8i SQL Reference* for more information on the syntax of `SET TRANSACTION` and `ALTER SESSION`.

Part IV

Optimizing Instance Performance

Part IV describes how to tune various elements of your database system to optimize performance of an Oracle instance.

The chapters are:

- Chapter 18, "Tuning CPU Resources"
- Chapter 19, "Tuning Memory Allocation"
- Chapter 20, "Tuning I/O"
- Chapter 21, "Tuning Resource Contention"
- Chapter 22, "Tuning Networks"
- Chapter 23, "Tuning the Operating System"
- Chapter 24, "Tuning Instance Recovery Performance"

18

Tuning CPU Resources

This chapter describes how to solve CPU resource problems.

This chapter contains the following sections:

- Understanding CPU Problems
- Detecting and Solving CPU Problems
- Solving CPU Problems by Changing System Architectures

Understanding CPU Problems

To address CPU problems, first establish appropriate expectations for the amount of CPU resources your system should be using. Then, determine whether sufficient CPU resources are available, and recognize when your system is consuming too many resources. Begin by determining the amount of CPU resources the Oracle instance utilizes with your system in the following three cases:

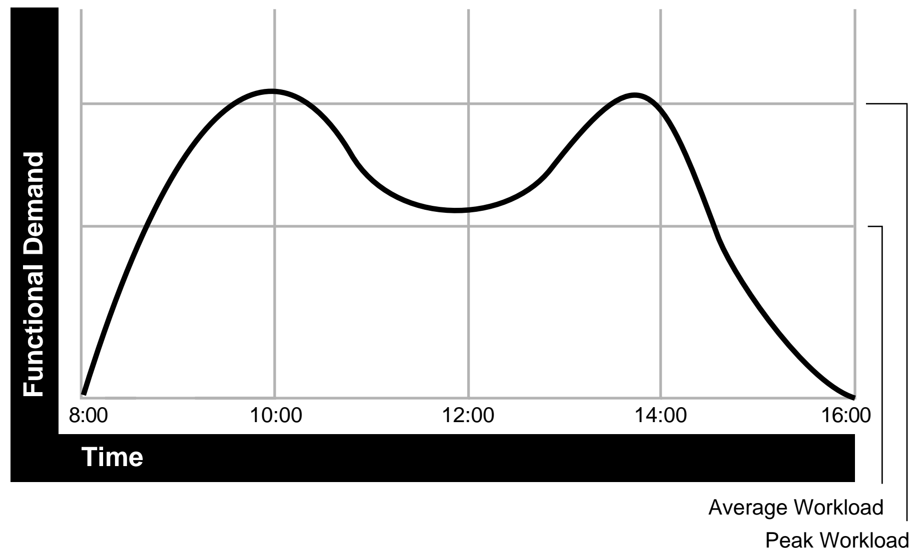
- System is idle (when little Oracle and non-Oracle activity exists)
- System at average workloads
- System at peak workloads

You can capture various workload snapshots using the `UTLBSTAT/UTLESTAT` utility, found in the `ORACLE_HOME/rdbms/admin/` directory on UNIX and in the `ORACLE_HOME/rdbms81/admin` directory on NT. Operating system tools, such as `vmstat`, `sar`, and `iostat` on UNIX and Performance Monitor on NT, should be run during the same time interval as `UTLBSTAT/UTLESTAT` to provide a complimentary view of the overall statistics.

Note: Release 8.1.6 also contains a new package called `STATSPACK` that improves on the `UTLBSTAT/UTLESTAT` process. For more information, see "Supported Scripts" on page 11-7.

Workload is an important factor when evaluating your system's level of CPU utilization. During peak workload hours, 90% CPU utilization with 10% idle and waiting time may be acceptable. Even 30% utilization at a time of low workload may be understandable. However, if your system shows high utilization at normal workload, then there is no room for a peak workload. For example, Figure 18–1 illustrates workload over time for an application having peak periods at 10:00 AM and 2:00 PM.

Figure 18–1 Average Workload and Peak Workload



This example application has 100 users working 8 hours a day, for a total of 800 hours per day. Each user entering one transaction every 5 minutes translates into 9,600 transactions daily. Over an 8-hour period, the system must support 1,200 transactions per hour, which is an average of 20 transactions per minute. If the demand rate were constant, then you could build a system to meet this average workload.

However, usage patterns are not constant—and in this context, 20 transactions per minute can be understood as merely a minimum requirement. If the peak rate you need to achieve is 120 transactions per minute, then you must configure a system that can support this peak workload.

For this example, assume that at peak workload, Oracle uses 90% of the CPU resource. For a period of average workload, then, Oracle uses no more than about 15% of the available CPU resource, as illustrated in the following equation:

*20 tpm/120 tpm * 90% = 15%*

Where *tpm* is transactions per minute.

If the system requires 50% of the CPU resource to achieve 20 tpm, then a problem exists: the system cannot achieve 120 transactions per minute using 90% of the CPU. However, if you tuned this system so that it achieves 20 tpm using only 15% of the

CPU, then, assuming linear scalability, the system might achieve 120 transactions per minute using 90% of the CPU resources.

As users are added to an application, the workload can rise to what had previously been peak levels. No further CPU capacity is then available for the new peak rate, which is actually higher than the previous.

CPU capacity issues can be addressed with the following:

1. Tuning; that is, detecting and solving CPU problems from excessive:
   - System CPU Utilization
   - Oracle CPU Utilization
2. Increasing hardware capacity, including changing the system architecture.

   See Also: For more information about improving your system architecture, see Chapter 2, "Performance Tuning Methods".

3. Reducing the impact of peak load use patterns by prioritizing CPU resource allocation. Oracle's Database Resource Manager does this by allocating and managing CPU resources among database users and applications.

   See Also: For more information about Oracle's Database Resource Manager, see *Oracle8i Concepts* and *Oracle8i Administrator's Guide.*

Detecting and Solving CPU Problems

If you suspect a problem with CPU usage, check two areas:

- System CPU Utilization
- Oracle CPU Utilization

System CPU Utilization

Oracle statistics report CPU use by Oracle sessions only, whereas every process running on your system affects the available CPU resources. Therefore, tuning non-Oracle factors can also improve Oracle performance.

Use operating system monitoring tools to determine what processes are running on the system as a whole. If the system is too heavily loaded, check the memory, I/O, and process management areas described later in this section.

Tools such as `sar -u` on many UNIX-based systems let you examine the level of CPU utilization on your entire system. CPU utilization in UNIX is described in statistics that show user time, system time, idle time, and time waiting for I/O. A CPU problem exists if idle time and time waiting for I/O are both close to zero (less than 5%) at a normal or low workload.

On NT, use Performance Monitor to examine CPU utilization. Performance Manager provides statistics on processor time, user time, privileged time, interrupt time, and DPC time. (NT Performance Monitor is not the same as Performance Manager, which is an Oracle Enterprise Manager tool.)

Note: This section describes how to check system CPU utilization on most UNIX-based and NT systems. For other platforms, see your operating system documentation.

Memory Management

Check the following memory management areas:

Paging and Swapping Use tools such as `sar` or `vmstat` on UNIX or Performance Monitor on NT to investigate the cause of paging and swapping.

Oversize Page Tables On UNIX, if the processing space becomes too large, then it may result in the page tables becoming too large. This is not an issue on NT.

I/O Management

Check the following I/O management issues:

Thrashing Ensure that your workload fits into memory, so the machine is not thrashing (swapping and paging processes in and out of memory). The operating system allocates fixed portions of time during which CPU resources are available to your process. If the process wastes a large portion of each time period checking to be sure that it can run and ensuring that all necessary components are in the machine, then the process may be using only 50% of the time allotted to actually perform work.

Client/Server Round Trips The latency of sending a message may result in CPU overload. An application often generates messages that need to be sent through the network over and over again, resulting in significant overhead before the message is actually sent. To alleviate this problem, batch the messages and perform the

overhead only once, or reduce the amount of work. For example, you can use array inserts, array fetches, and so on.

See Also: For more details on tuning I/O, see Chapter 20, "Tuning I/O".

Process Management

Check the following process management issues:

Scheduling and Switching The operating system may spend excessive time scheduling and switching processes. Examine the way in which you are using the operating system, because you could be using too many processes. On NT systems, do not overload your server with too many non-Oracle processes.

Context Switching Due to operating system specific characteristics, your system could be spending a lot of time in context switches. Context switching can be expensive, especially with a large SGA. Context switching is not an issue on NT, which has only one process per instance. All threads share the same page table.

Programmers often create single-purpose processes, exit the process, and create a new one. Doing this re-creates and destroys the process each time. Such logic uses excessive amounts of CPU, especially with applications that have large SGAs. This is because you need to build the page tables each time. The problem is aggravated when you pin or lock shared memory, because you have to access every page.

For example, if you have a 1 gigabyte SGA, then you may have page table entries for every 4K, and a page table entry may be 8 bytes. You could end up with (1G/4K) * 8B entries. This becomes expensive, because you need to continually make sure that the page table is loaded.

Parallel execution and the multi-threaded server become areas of concern if `MINSERVICE` has been set too low (set to 10, for example, when you need 20). For an application that is performing small lookups, this may not be wise. In this situation, it becomes inefficient for both the application and the system.

Oracle CPU Utilization

This section explains how to examine the processes running in Oracle. Three dynamic performance views provide information on Oracle processes:

- `V$SYSSTAT` shows Oracle CPU usage for all sessions. The statistic "CPU used by this session" shows the aggregate CPU used by all sessions.

- `V$SESSTAT` shows Oracle CPU usage per session. You can use this view to determine which particular session is using the most CPU.
- `V$RSRC_CONSUMER_GROUP` shows CPU utilization statistics on a per consumer group basis, if you are running the Oracle Database Resource Manager.

For example, if you have 8 CPUs, then for any given minute in real time, you have 8 minutes of CPU time available. On NT and UNIX, this can be either user time or time in system mode (*privileged* mode on NT). If your process is not running, then it is waiting. Thus, CPU time utilized by all systems may be greater than one minute per interval.

At any given moment, you know how much time Oracle has used on the system. So, if 8 minutes are available and Oracle uses 4 minutes of that time, then you know that 50% of all CPU time is used by Oracle. If your process is not consuming that time, then some other process is. You then need to identify the processes that are using CPU time. If you can, determine why the processes use so much CPU time and attempt to tune them. Possible areas to research include, but are not limited to, the following:

- Reparsing SQL Statements
- Read Consistency
- Scalability Limitations Within the Application
- Wait Detection
- Latch Contention

Reparsing SQL Statements

When Oracle executes a SQL statement, it parses it to determine whether the syntax and its contents are correct. This process can consume significant overhead. Once parsed, Oracle does not parse the statement again unless the parsing information is aged from the memory cache and is no longer available. Ineffective memory sharing among SQL statements can result in reparsing. Use the following procedure to determine whether reparsing is occurring:

- Get the parse time CPU and CPU figures used by this session from the "Statistics" section of the `estat` report or from `V$SYSTATS`. For example:

```
SELECT * FROM V$SYSSTAT
WHERE NAME IN('parse time cpu', 'parse time elapsed', 'parse count (hard)');
```

  Now you can detect the general response time on parsing. The more your application is parsing, the more contention exists, and the more time your

system spends waiting. If parse time CPU represents a large percentage of the CPU time, then time is being spent parsing instead of executing statements. If this is the case, then it is likely that the application is using literal SQL and not sharing it, or the shared pool is poorly configured.

- Query V$SQLAREA to find frequently reparsed statements. For example:

```
SELECT SQL_TEXT, PARSE_CALLS, EXECUTIONS
FROM V$SQLAREA
ORDER BY PARSE_CALLS;
```

Tune the statements with the higher numbers of parse calls.

If the parse time CPU is only a small percentage of the total CPU used, then you should determine where the CPU resources are going. There are several things you can do to help with this.

1. Find statements with large numbers of buffer gets, because these are typically heavy on CPU.

   The following statement finds SQL statements which frequently access database buffers. Such statements are probably looking at many rows of data.

   ```
   SELECT ADDRESS, HASH_VALUE, BUFFER_GETS, EXECUTIONS,
    BUFFER GETS/EXECUTIONS "GETS/EXEC", SQL_TEXT
   FROM V$SQLAREA
   WHERE BUFFER_GETS > 50000
       AND EXECUTIONS > 0
   ORDER BY 3;
   ```

   This example shows which SQL statements have the most buffer_gets and use the most CPU. The statements of interest are those with a large number of gets per execution, especially if execution is high. It is very beneficial to have an understanding of the application components to know which statements are expected to be expensive.

   Note: The 50000 cut-off value is an arbitrary starting point, and it should be increased or decreased gradually until the top 10 to 20 statements are listed. This statement does not highlight CPU-intensive PL/SQL blocks.

2. After candidate statements have been isolated, the full statement text can be obtained using the following query, substituting relevant values for ADDRESS and HASH_VALUE pairs. For example:

```
SELECT SQL_TEXT
FROM V$SQLTEXT
WHERE ADDRESS='&ADDRESS_WANTED'
    AND HASH_VALUe=&HASH_VALUE
ORDER BY piece;
```

   The statement can then be explained (using EXPLAIN PLAN) or isolated for further testing to see how CPU-intensive it really is. If the statement uses bind variables and if your data is highly skewed, then the statement may only be CPU-intensive for certain bind values.

3. Find which sessions are responsible for most CPU usage. The following statement helps locate sessions which have used the most CPU:

```
SELECT v.SID, SUBSTR(s.NAME,1,30) "Statistic", v.VALUE
FROM V$STATNAME s, V$SESSTAT v
WHERE s.NAME = 'CPU used by this session'
    AND v.STATISTIC# = s.STATISTIC#
    AND v.VALUE > 0
ORDER BY 3;
```

   Note: CPU time is cumulative; therefore, a session that has been connected for several days may appear to be heavier on CPU than one that has only been connected for a short period of time. Thus, it is better to write a script to sample the difference in the statistic between two known points in time, letting you see how much CPU was used by each session in a known time frame.

   After any CPU-intensive sessions have been identified, the V$SESSION view can be used to get more information. At this stage, it is generally best to revert to user session tracing (SQL_TRACE) to determine where the CPU is being used.

4. Trace typical user sessions using the SQL_TRACE option to see how CPU is apportioned amongst the main application statements.

   After these statements have been identified, you have the following three options for tuning them:

   * Rewrite the application so that statements do not continually reparse.

* Reduce parsing by using the initialization parameter SESSION_CACHED_CURSORS.
* If the parse count is small, the execute count is small, and the SQL statements are very similar except for the WHERE clause, then you may find that hard coded values are being used instead of bind variables. Use bind variables to reduce parsing.

Read Consistency

Your system may spend excessive time rolling back changes to blocks in order to maintain a consistent view. Consider the following scenarios:

- If there are many small transactions and an active long-running query is running in the background on the same table where the inserts are happening, then the query may need to roll back many changes.
- If the number of rollback segments is too small, then your system could also be spending a lot of time rolling back the transaction table. Your query may have started long ago; because the number of rollback segments and transaction tables is very small, your system frequently needs to reuse transaction slots.

  Note: The average wait time should be close to zero. (V$SYSSTAT also shows the average wait time per parse.)

  See Also: For information on approaches to SQL statement tuning, see Chapter 9, "Optimizing SQL Statements".

  A solution is to make more rollback segments, or to increase the commit rate. For example, if you batch ten transactions and commit them once, then you reduce the number of transactions by a factor of ten.

- If your system must scan too many buffers in the foreground to find a free buffer, then it wastes CPU resources. To alleviate this problem, tune the DBW*n* process(es) to write more frequently.

  You can also increase the size of the buffer cache to enable the database writer process(es) to keep up. To find the average number of buffers the system scans at the end of the least recently used list (LRU) to find a free buffer, use the following formula:

$$1 + \frac{\text{"free buffers inspected"}}{\text{"free buffers requested"}} = \text{avg. buffers scanned}$$

On average, you would expect to see 1 or 2 buffers scanned. If more than this number are being scanned, then increase the size of the buffer cache or tune the DBW*n* process(es).

Use the following formula to find the number of buffers that were dirty at the end of the LRU:

$$\frac{\text{"dirty buffers inspected"}}{\text{"free buffers inspected"}} = \text{dirty buffers}$$

If many dirty buffers exist, then possibly the DBW*n* process(es) cannot keep up. Again, increase the buffer cache size or tune the DBW*n* process.

> **Note:** Query the `V$SYSSTAT` view to find the values of "free buffers inspected" and "dirty buffers inspected".

Scalability Limitations Within the Application

In most of this CPU tuning discussion, we assume you can achieve linear scalability, but this is never actually the case. How flat or nonlinear the scalability is indicates how far away from optimal performance your system is. Problems in your application might be adversely affecting scalability. Examples of this include too many indexes, right-hand index problems, too much data in the blocks, or not properly partitioning the data. These types of contention problems waste CPU cycles and prevent the application from attaining linear scalability.

Wait Detection

Whenever an Oracle process waits for something, it records it as a *wait* using one of a set of predefined wait events. (See `V$EVENT_NAME` for a list of all wait events). Some of these events can be considered *idle* events; i.e., the process is waiting for work. Other events indicate time spent waiting for a resource or action to complete. By comparing the relative time spent waiting on each wait event and the "CPU used by this session" (from above), you can see where the Oracle instance is spending most of its time. To get an indication of where time is spent, follow these steps:

1. Review either the V$SYSTATS view or the wait events section of the UTLBSTAT/UTLESTAT report.
2. Ignore any idle wait events. Common idle wait events include:
   - Client message
   - SQL*Net message from client
   - SQL*Net more data from client
   - RDBMS IPC message
   - Pipe get
   - Null event
   - PMON timer
   - SMON timer
   - Parallel query dequeue
3. Ignore any wait events that represent a very small percentage of the total time waited.
4. Add the remaining wait event times, and calculate each one as a percentage of total time waited.
5. Compare the total time waited with the CPU used by this session figure.
6. Find the event with the largest wait event time. This may be the first item you want to tune.

Latch Contention

Latch contention is a symptom of CPU problems; it is not usually a cause. To resolve it, you must locate the latch contention within your application, identify its cause, and determine which part of your application is poorly written.

In some cases, the spin count may be set too high. It's also possible that one process may be holding a latch that another process is attempting to secure. The process attempting to secure the latch may be endlessly spinning. After a while, this process may go to sleep and later resume processing and repeat its ineffectual spinning. To resolve this:

- Check the Oracle latch statistics. The "latch free" event in V$SYSTEM_EVENT shows how long processes have been waiting for latches. If there is no latch contention, then this statistic does not appear. If there is a lot of contention, then

it may be better for a process to go to sleep at once when it cannot obtain a latch, rather than use CPU time by spinning.

- Look for the ratio of CPUs to processes. If there are large numbers of both, then many processes can run. But, if a single process is holding a latch on a system with ten CPUs, then reschedule that process so it is not running. But, ten other processes may run ineffectively trying to secure the same latch. This situation wastes, in parallel, some CPU resource.
- Check `V$LATCH_MISSES`, which indicates where in the Oracle code most contention occurs.

> **Note:** Tuning the `SPIN_COUNT` actually tunes the symptom and not the real problem. Furthermore, setting `SPIN_COUNT` may actually increase CPU waits. If a post-wait driver is available on the system, then evaluate its implementation.

Solving CPU Problems by Changing System Architectures

If you have maximized the CPU power on your system and have exhausted all means of tuning your system's CPU use, then consider redesigning your system on another architecture. Moving to a different architecture might improve CPU use. This section describes architectures you could consider using. This section contains the following possibilities:

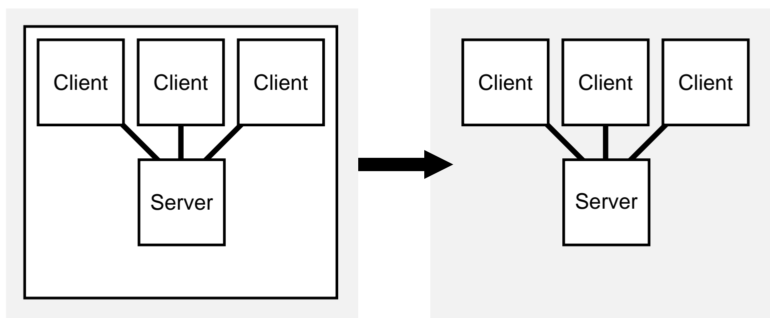
- Single Tier to Two-Tier
- Multi-Tier: Using Smaller Client Machines
- Two-Tier to Three-Tier
- Three-Tier
- Oracle Parallel Server

> **Note:** If you are running a multi-tier system, then check all levels for CPU utilization. For example, on a three-tier system, your server might be mostly idle while your second tier is completely busy. To resolve this, tune the second tier rather than the server or the third tier. In a multi-tier system, it is usually not the server that has a performance problem. It is usually the clients and the middle tier.

Single Tier to Two-Tier

Consider whether changing from several clients with one server, all running on a single machine (single tier), to a two-tier client/server configuration would relieve CPU problems.

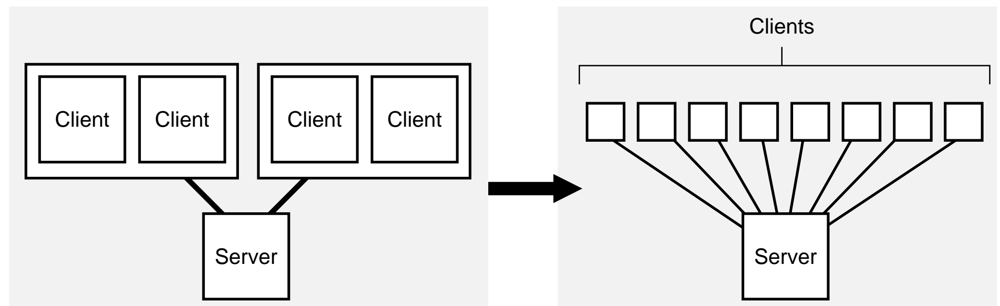
Figure 18–2 Single Tier to Two-Tier



Multi-Tier: Using Smaller Client Machines

Consider whether using smaller clients improves CPU usage rather than using multiple clients on larger machines. This strategy may be helpful with either two-tier or three-tier configurations.

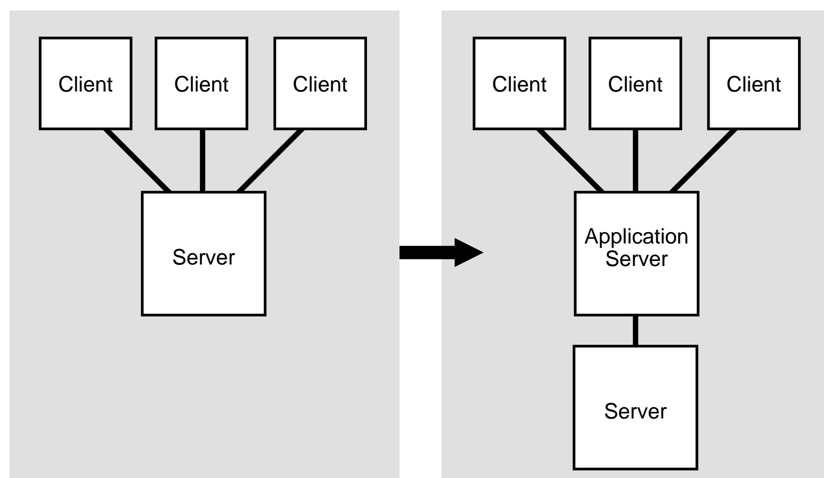
Figure 18–3 Multi-Tier Using Smaller Clients



Two-Tier to Three-Tier

If your system runs with multiple layers, then consider whether moving from a two-tier to three-tier configuration and introducing an application server or a transaction processing monitor might be a good solution.

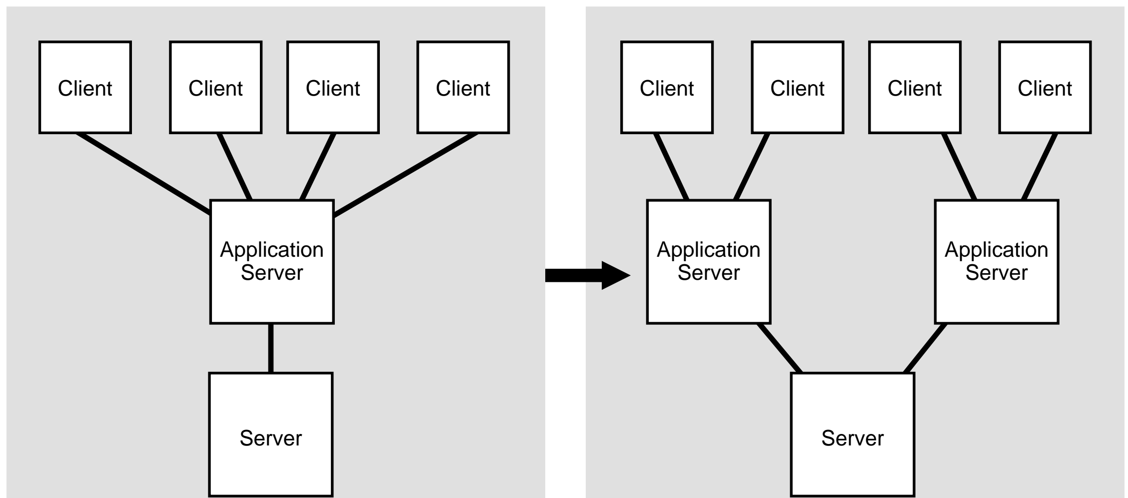
Figure 18–4 Two-Tier to Three-Tier



Three-Tier

Consider using one or more application servers or multiple transaction processing monitors.

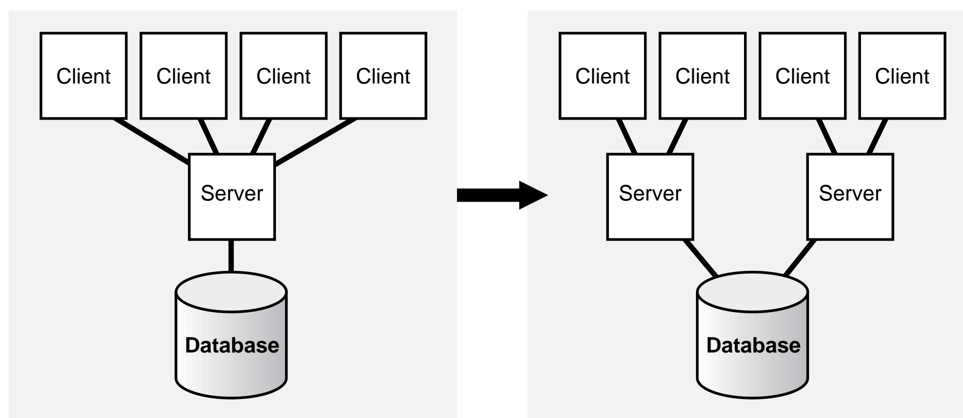
Figure 18–5 Three-Tier with Multiple Application Servers



Oracle Parallel Server

Consider whether incorporating Oracle Parallel Server would solve your CPU problems.

Figure 18–6 Oracle Parallel Server



19

Tuning Memory Allocation

This chapter explains how to allocate memory to database structures. Proper sizing of these structures greatly improves database performance.

This chapter contains the following sections:

- Understanding Memory Allocation Issues
- Detecting Memory Allocation Problems
- Solving Memory Allocation Problems

Understanding Memory Allocation Issues

Oracle stores information in memory and on disk. Memory access is much faster than disk access; therefore, it is better for data requests to be satisfied by access to memory instead of by access to disk. For best performance, store as much data as possible in memory. However, memory resources on your operating system are likely to be limited. Tuning memory allocation involves distributing available memory to Oracle memory structures.

Oracle's memory requirements depend on your application. Therefore, tune memory allocation *after* tuning your application and SQL statements. If you allocate memory before tuning your application and SQL statements, then you may need to resize some Oracle memory structures to meet the needs of your modified statements and application.

Also, tune memory allocation *before* you tune I/O. Allocating memory establishes the amount of I/O necessary for Oracle to operate. This chapter shows you how to allocate memory to perform as little I/O as possible.

The following terms are used in this discussion:

block	A unit of data transfer between main memory and disk. Many blocks from one section of memory address space form a segment.
buffer	A main memory address in which the buffer manager caches currently and recently used data read from disk. Over time, a buffer may hold different blocks. When a new block is needed, the buffer manager may discard an old block and replace it with a new one.
buffer pool	A collection of buffers.
cache or buffer cache	All buffers and buffer pools.
segment	A set of extents allocated for a specific type of database object such as a table, index, or cluster.

See Also: For information on how to perform I/O as efficiently as possible, see Chapter 20, "Tuning I/O".

Detecting Memory Allocation Problems

When you use operating system tools to examine the size of Oracle processes, such as `ps -efl` or `ps -aux` on UNIX, you may notice that the processes seem large. To interpret the statistics shown, determine how much of the process size is attributable to shared memory, heap, and executable stack, and how much is the actual amount of memory the given process consumes.

The SZ statistic is given in units of page size (normally 4KB), and it normally includes the shared overhead. To calculate the private, or per-process memory usage, subtract shared memory and executable stack figures from the value of SZ. For example:

SZ	+20,000
minus SHM	- 15,000
minus EXECUTABLE	- 1,000
actual per-process memory	4,000

In this example, the individual process consumes only 4,000 pages; the other 16,000 pages are shared by all processes.

> **See Also:** *Oracle for UNIX Performance Tuning Tips* or your operating system documentation.

Solving Memory Allocation Problems

The rest of this chapter explains how to tune memory allocation. For best results, resolve memory issues in the order presented here:

1. Tuning Operating System Memory Requirements
2. Tuning the Redo Log Buffer
3. Tuning Private SQL and PL/SQL Areas
4. Tuning the Shared Pool
5. Tuning the Buffer Cache
6. Tuning Multiple Buffer Pools
7. Tuning Sort Areas
8. Reallocating Memory
9. Reducing Total Memory Usage

Tuning Operating System Memory Requirements

Begin tuning memory allocation by tuning your operating system with these goals:

- Reducing Paging and Swapping
- Fitting the System Global Area into Main Memory
- Allocating Adequate Memory to Individual Users

These goals apply in general to most operating systems, but the details of operating system tuning vary.

> **See Also:** For more information on tuning operating system memory usage, see your operating system hardware and software documentation, as well as your Oracle operating system-specific documentation.

Reducing Paging and Swapping

Your operating system may store information in these places:

- Real memory
- Virtual memory
- Expanded storage
- Disk

The operating system may also move information from one storage location to another. This process is known as *paging* or *swapping*. Many operating systems page and swap to accommodate large amounts of information that do not fit into real memory. However, excessive paging or swapping can reduce the performance of many operating systems.

Monitor your operating system behavior with operating system utilities. Excessive paging or swapping indicates that new information is often being moved into memory. In this case, your system's total memory may not be large enough to hold everything for which you have allocated memory. Either increase the total memory on your system or decrease the amount of memory allocated.

Fitting the System Global Area into Main Memory

Because the purpose of the System Global Area (SGA) is to store data in memory for fast access, the SGA should always be within main memory. If pages of the SGA are swapped to disk, then its data is no longer quickly accessible. On most operating

systems, the disadvantage of excessive paging significantly outweighs the advantage of a large SGA.

Although it is best to keep the entire SGA in memory, the contents of the SGA are split logically between *hot* and *cold* parts. The hot parts are always in memory, because they are always being referenced. Some cold parts may be paged out, and a performance penalty may result from bringing them back in. A performance problem likely occurs, however, when the hot part of the SGA cannot remain in memory.

Data is swapped to disk because it is not being referenced. You can cause Oracle to read the entire SGA into memory when you start your instance by setting the value of the initialization parameter `PRE_PAGE_SGA` to `YES`. Operating system page table entries are then pre-built for each page of the SGA. This setting may increase the amount of time necessary for instance startup, but it is likely to decrease the amount of time necessary for Oracle to reach its full performance capacity after startup.

Note: This setting does not prevent your operating system from paging or swapping the SGA after it is initially read into memory.

`PRE_PAGE_SGA` may increase the process startup duration, because every process that starts must attach itself to the SGA. The cost of this strategy is fixed; however, you may simply determine that 20,000 pages must be touched every time a process starts. This approach may be useful with some applications, but not with all applications. Overhead may be significant if your system frequently creates and destroys processes by, for example, continually logging on and logging off.

The advantage that `PRE_PAGE_SGA` can afford depends on page size. For example, if the SGA is 80MB in size, and the page size is 4KB, then 20,000 pages must be touched to refresh the SGA (80,000/4 = 20,000).

If the system permits you to set a 4MB page size, then only 20 pages must be touched to refresh the SGA (80,000/4,000 = 20). The page size is operating system-specific and generally cannot be changed. Some operating systems, however, have a special implementation for shared memory whereby you can change the page size.

You can see how much memory is allocated to the SGA and each of its internal structures by issuing the following SQL statement:

```
SHOW SGA
```

The output of this statement could look like the following:

```
Total System Global Area                    18847360 bytes
Fixed Size                                     63104 bytes
Variable Size                               14155776 bytes
Database Buffers                             4096000 bytes
Redo Buffers                                  532480 bytes
```

Some IBM mainframe computer operating systems have expanded storage or special memory, in addition to main memory, to which paging can be performed very quickly. These operating systems may be able to page data between main memory and expanded storage faster than Oracle can read and write data between the SGA and disk. For this reason, allowing a larger SGA to be swapped may lead to better performance than ensuring that a smaller SGA remains in main memory. If your operating system has expanded storage, then take advantage of it by allocating a larger SGA despite the resulting paging.

Allocating Adequate Memory to Individual Users

On some operating systems, you may have control over the amount of physical memory allocated to each user. Be sure that all users are allocated enough memory to accommodate the resources they need to use their application with Oracle.

Depending on your operating system, these resources may include:

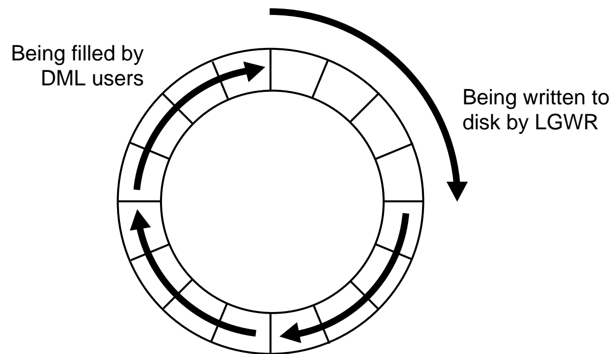
- The Oracle executable image
- The SGA
- Oracle application tools
- Application-specific data

On some operating systems, Oracle software can be installed so that a single executable image can be shared by many users. By sharing executable images among users, you can reduce the amount of memory required by each user.

Tuning the Redo Log Buffer

The LOG_BUFFER parameter reserves space for the redo log buffer that is fixed in size. On machines with fast processors and relatively slow disks, the processors may be filling the rest of the buffer in the time it takes the redo log writer to move a portion of the buffer to disk. The log writer process (LGWR) always starts when the buffer begins to fill. For this reason, a larger buffer makes it less likely that new entries collide with the part of the buffer still being written.

Figure 19–1 Redo Log Buffer



The log buffer is normally small compared with the total SGA size, and a modest increase can significantly enhance throughput.

Detecting Contention for Space in the Redo Log Buffer

When LGWR writes redo entries from the redo log buffer to a redo log file or disk, user processes can then copy new entries over the entries in memory 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.

The statistic `REDO BUFFER ALLOCATION RETRIES` reflects the number of times a user process waits for space in the redo log buffer. This statistic is available through the dynamic performance view `V$SYSSTAT`. By default, this view is available only to the user `SYS` and to users granted `SELECT ANY TABLE` system privilege, such as `SYSTEM`.

Use the following query to monitor these statistics over a period of time while your application is running:

```
SELECT NAME, VALUE
FROM V$SYSSTAT
'WHERE NAME = 'REDO BUFFER ALLOCATION RETRIES';
```

The information in V$SYSSTAT can also be obtained through the Simple Network Management Protocol (SNMP).

The value of REDO BUFFER ALLOCATION RETRIES should be near zero. If this value increments consistently, then processes have had to wait for space in the buffer. The wait may be caused by the log buffer being too small or by checkpointing. Increase the size of the redo log buffer, if necessary, by changing the value of the initialization parameter LOG_BUFFER. The value of this parameter, expressed in bytes, must be a multiple of DB_BLOCK_SIZE. Alternatively, improve the checkpointing or archiving process.

Note: Multiple archiver processes are not recommended. A single automatic ARCH process can archive redo logs, keeping pace with the LGWR process.

Tuning Private SQL and PL/SQL Areas

This section explains how to tune private SQL and PL/SQL areas in the following ways:

- Identifying Unnecessary Parse Calls
- Reducing Unnecessary Parse Calls

A trade-off exists between memory and reparsing. With significant amounts of reparsing, less memory is needed. If you reduce reparsing by creating more SQL statements, then client memory requirements increase. This is due to an increase in the number of open cursors.

Tuning private SQL areas entails identifying unnecessary parse calls made by your application and then reducing them. To reduce parse calls, you may need to increase the number of private SQL areas that your application can have allocated at once. Throughout this section, information about private SQL areas and SQL statements also applies to private PL/SQL areas and PL/SQL blocks.

Identifying Unnecessary Parse Calls

This section describes three techniques for identifying unnecessary parse calls.

Technique 1 Run your application with the SQL trace facility enabled. For each SQL statement in the trace output, the "count" statistic for the Parse step tells you how many times your application makes a parse call for the statement. This statistic includes parse calls satisfied by access to the library cache, as well as parse calls resulting in actually parsing the statement.

Note: This statistic does not include implicit parsing that occurs when an application executes a statement whose shared SQL area is no longer in the library cache. For information on detecting implicit parsing, see "Examining Library Cache Activity" on page 19-14.

If the count value for the Parse step is near the count value for the Execute step for a statement, then your application may be deliberately making a parse call each time it executes the statement. Try to reduce these parse calls through your application tool.

Technique 2 Another way to identify unnecessary parse calls is to check the V$SQLAREA view. Enter the following query:

```
SELECT SQL_TEXT, PARSE_CALLS, EXECUTIONS
FROM V$SQLAREA;
```

When the PARSE_CALLS value is close to the EXECUTION value for a given statement, you may be continually reparsing that statement.

Technique 3 You can also identify unnecessary parse calls by identifying the session in which they occur. It may be that particular batch programs or certain types of applications do most of the reparsing. To do this, execute the following query:

```
SELECT * FROM V$STATNAME
WHERE NAME IN ('parsecount (hard)','executecount');
```

Oracle responds with something similar to:

```
STATISTIC#,   NAME
------------  ---------
100           parsecount
90            executecount
```

Then, run a query similar to the following:

```
SELECT * FROM V$SESSTAT
WHERE STATISTICS# IN (90,100)
ORDER BY VALUE, SID;
```

The result is a list of all sessions and the amount of reparsing they do. For each system identifier (SID), go to `V$SESSION` to find the name of the program that causes the reparsing.

Reducing Unnecessary Parse Calls

Depending on the Oracle application tool you are using, you may be able to control how frequently your application performs parse calls and allocates and deallocates private SQL areas. Whether your application reuses private SQL areas for multiple SQL statements determines how many parse calls your application performs and how many private SQL areas the application requires.

In general, an application that reuses private SQL areas for multiple SQL statements does not need as many private SQL areas as an application that does not reuse private SQL areas. However, an application that reuses private SQL areas must perform more parse calls, because the application must make a new parse call whenever an existing private SQL area is reused for a new SQL statement.

Be sure that your application can open enough private SQL areas to accommodate all your SQL statements. If you allocate more private SQL areas, then you may need to increase the limit on the number of cursors permitted for a session. You can increase this limit by increasing the value of the initialization parameter `OPEN_CURSORS`. The default value for OPEN_CURSORS is 50, and the range is from 1 to `UB4MAXVAL`.

The ways in which you control parse calls and allocation and deallocation of private SQL areas depends on your Oracle application tool. The following sections introduce the methods used for some tools. These methods apply only to private SQL areas and not to shared SQL areas.

Reducing Parse Calls with the Oracle Precompilers When using the Oracle precompilers, you can control private SQL areas and parse calls by setting three clauses. In Oracle mode, the clauses and their defaults are as follows:

- `HOLD_CURSOR` = yes
- `RELEASE_CURSOR` = no
- `MAXOPENCURSORS` = *desired value*

Oracle recommends that you *not* use ANSI mode, in which the values of HOLD_CURSOR and RELEASE_CURSOR are switched.

The precompiler clauses can be specified in two ways:

- On the precompiler command line
- Within the precompiler program

With these clauses, you can employ different strategies for managing private SQL areas during the course of the program.

> **See Also:** For more information on these calls, see *Pro*C/C++ Precompiler Programmer's Guide*.

Reducing Parse Calls with Oracle Forms With Oracle Forms, you also have some control over whether your application reuses private SQL areas. You can exercise this control in three places:

- At the trigger level
- At the form level
- At run time

> **See Also:** For more information on the reuse of private SQL areas by Oracle Forms, see the *Oracle Forms Reference* manual.

Tuning the Shared Pool

The shared pool contains the library cache of shared SQL requests, the dictionary cache, stored procedures, and other cache structures that are specific to a particular instance configuration. For example, in a multi-threaded server (MTS) configuration, the session and private SQL area for each client process is included in the shared pool. When the instance is configured for parallel execution, the shared pool includes the parallel execution message buffers.

Proper sizing of the shared pool can reduce resource consumption in at least three ways:

1. Parse time is avoided if the SQL statement is already in the shared pool. This saves CPU resources.
2. Application memory overhead is reduced, because all applications use the same pool of shared SQL statements and dictionary resources.

3. I/O resources are saved, because dictionary elements which are in the shared pool do not require disk access.

Managing Data in the Shared Pool

The algorithm Oracle uses to manage data in the shared pool tends to hold dictionary cache data in memory longer than library cache data. Therefore, tuning the library cache to an acceptable cache hit ratio often ensures that the data dictionary cache hit ratio is also acceptable. Allocating space in the shared pool for session information is necessary only if you are using MTS architecture.

In the shared pool, some of the caches are dynamic—their sizes automatically increase or decrease as needed. These dynamic caches include the library cache and the data dictionary cache. Objects are aged out of these caches if the shared pool runs out of room. For this reason you may need to increase the shared pool size if the frequently used set of data does not fit within it. A cache miss on the data dictionary cache or library cache is more expensive than a miss on the buffer cache. For this reason, allocate sufficient memory to the shared pool before allocating to the buffer cache.

For most applications, shared pool size is critical to Oracle performance. (Shared pool size is less important only for applications that issue a very limited number of discrete SQL statements.) The shared pool holds both the data dictionary cache and the fully parsed or compiled representations of PL/SQL blocks and SQL statements. PL/SQL blocks include procedures, functions, packages, triggers, and any anonymous PL/SQL blocks submitted by client programs.

If the shared pool is too small, then the server must dedicate resources to managing the limited amount of available space. This consumes CPU resources and causes contention, because Oracle imposes restrictions on the parallel management of the various caches. The more you use triggers and stored procedures, the larger the shared pool must be. It may even reach a size measured in hundreds of megabytes.

Because it is better to measure statistics during a confined period than from startup, you can determine the library cache and row cache (data dictionary cache) hit ratios from the following queries. The results show the miss rates for the library cache and row cache. In general, the number of reparses reflects the library cache. If the ratios are close to 1, then you do not need to increase the pool size.

```
SELECT (SUM(PINS - RELOADS)) / SUM(PINS) "LIB CACHE"
FROM V$LIBRARYCACHE;

SELECT (SUM(GETS - GETMISSES - USAGE - FIXED)) / SUM(GETS) "ROW CACHE"
FROM V$ROWCACHE;
```

The amount of free memory in the shared pool is reported in V$SGASTAT. Report the current value from this view using the following query:

```
SELECT * FROM V$SGASTAT WHERE NAME = 'FREE MEMORY';
```

If there is always free memory available within the shared pool, then increasing the size of the pool offers little or no benefit. However, just because the shared pool is full does not necessarily mean there is a problem.

After an entry has been loaded into the shared pool, it cannot be moved. As more entries are loaded, the free memory becomes discontiguous, and the shared pool may become fragmented.

You can use the PL/SQL package DBMS_SHARED_POOL, located in dbmspool.sql, to manage the shared pool. The comments in the code describe how to use the procedures within the package.

> **See Also:** For more information about DBMS_SHARED_POOL, see the *Oracle8i Supplied PL/SQL Packages Reference.*

Loading PL/SQL Objects into the Shared Pool Oracle loads objects into the shared pool using *pages* that are 4KB in size. These pages load chunks of segmented PL/SQL code. The pages do not need to be contiguous. Therefore, Oracle does not need to allocate large sections of contiguous memory for loading objects into the shared pool. This reduces the need for contiguous memory and improves performance. However, Oracle loads all of a package if any part of the package is called.

Depending on user needs, it may or may not be prudent to pin packages in the shared pool. Nonetheless, Oracle recommends pinning, especially for frequently used application objects.

> **See Also:** For information on how to pin packages with the DBMS_SHARED_POOL package, see Chapter 13, "Managing Shared SQL and PL/SQL Areas" and *Oracle8i Supplied PL/SQL Packages Reference.*

Library Cache and Row Cache Hit Ratios Library cache and row cache hit ratios are important. If free memory is near zero, and if either the library cache hit ratio or the row cache hit ratio is less than 0.95, then increase the size of the shared pool until the ratios stop improving.

The following sections explains how to allocate memory for key memory structures of the shared pool. Structures are listed in order of importance for tuning.

- Tuning the Library Cache
- Tuning the Data Dictionary Cache
- Tuning the Large Pool and Shared Pool for the MTS Architecture
- Tuning Reserved Space from the Shared Pool

> **Note:** If you are using a reserved size for the shared pool, then see "SHARED_POOL_SIZE Too Small" on page 19-27.

Tuning the Library Cache

The library cache holds executable forms of SQL cursors, PL/SQL programs, and JAVA classes. It also caches descriptive information, or metadata, about schema objects. Oracle uses this metadata when parsing SQL cursors or during the compilation of PL/SQL programs. The latter type of memory is seldom a concern for performance, so this section focuses on tuning as it relates to cursors, PL/SQL programs, and JAVA classes. These are collectively referred to as *application logic*.

Examining Library Cache Activity Library cache misses can occur on either the parse or the execute step in the processing of a SQL statement.

If an application makes a *parse* call for a SQL statement, and if the parsed representation of the statement does not already exist in a shared SQL area in the library cache, then Oracle parses the statement and allocates a shared SQL area. You may be able to reduce library cache misses on parse calls by ensuring that SQL statements can share a shared SQL area whenever possible.

If an application makes an *execute* call for a SQL statement, and if the shared SQL area containing the parsed representation of the statement has been deallocated from the library cache to make room for another statement, then Oracle implicitly reparses the statement, allocates a new shared SQL area for it, and executes it. You may be able to reduce library cache misses on execution calls by allocating more memory to the library cache.

You can monitor statistics reflecting library cache activity by examining the dynamic performance view `V$LIBRARYCACHE`. These statistics reflect all library cache activity since the most recent instance startup. By default, this view is available only to the user `SYS` and to users granted `SELECT ANY TABLE` system privilege, such as `SYSTEM`.

Each row in this view contains statistics for one type of item kept in the library cache. The item described by each row is identified by the value of the `NAMESPACE`

column. Rows of the table with the following NAMESPACE values reflect library cache activity for SQL statements and PL/SQL blocks:

- SQL AREA
- TABLE/PROCEDURE
- BODY
- TRIGGER

Rows with other NAMESPACE values reflect library cache activity for object definitions that Oracle uses for dependency maintenance.

These columns of the V$LIBRARYCACHE table reflect library cache misses on execution calls:

PINS	Shows the number of times an item in the library cache was executed.
RELOADS	Shows the number of library cache misses on execution steps.

Monitor the statistics in the V$LIBRARYCACHE table over a period of time with the following query:

```
SELECT SUM(PINS) "EXECUTIONS",
SUM(RELOADS) "CACHE MISSES WHILE EXECUTING"
FROM V$LIBRARYCACHE;
```

The output of this query could look like the following:

```
EXECUTIONS CACHE MISSES WHILE EXECUTING
---------- ----------------------------
320871                 549
```

Examining the data returned by the sample query leads to these observations:

- The sum of the EXECUTIONS column indicates that SQL statements, PL/SQL blocks, and object definitions were accessed for execution a total of 320,871 times.
- The sum of the CACHE MISSES WHILE EXECUTING column indicates that 549 of those executions resulted in library cache misses causing Oracle to implicitly reparse a statement or block or reload an object definition because it aged out of the library cache.
- The ratio of the total misses to total executions is about 0.17%. This value means that only 0.17% of executions resulted in reparsing.

Total misses should be near 0. If the ratio of misses to executions is more than 1%, then try to reduce the library cache misses through the means discussed in the next section.

You can reduce library cache misses by:

- Allocating Additional Memory for the Library Cache
- Writing Similar SQL Statements

Allocating Additional Memory for the Library Cache To ensure that shared SQL areas remain in the cache after their SQL statements are parsed, increase the amount of memory available to the library cache until the V$LIBRARYCACHE.RELOADS value is near 0. To increase the amount of memory available to the library cache, increase the value of the initialization parameter SHARED_POOL_SIZE. The maximum value for this parameter depends on your operating system. This measure reduces implicit reparsing of SQL statements and PL/SQL blocks on execution.

To take advantage of additional memory available for shared SQL areas, you may also need to increase the number of cursors permitted for a session. You can do this by increasing the value of the initialization parameter OPEN_CURSORS.

Be careful not to induce paging and swapping by allocating too much memory for the library cache. The benefits of a library cache large enough to avoid cache misses can be partially offset by reading shared SQL areas into memory from disk whenever you need to access them.

> **See Also:** For more information, see "SHARED_POOL_SIZE Too Small" on page 19-27.

Writing Similar SQL Statements You may be able to reduce library cache misses on parse calls by ensuring that SQL statements and PL/SQL blocks use a shared SQL area whenever possible. Two separate occurrences of a SQL statement or PL/SQL block can use a shared SQL area if they follow these criteria:

- The text of the SQL statements or PL/SQL blocks must be identical, character for character, including spaces and case.

  The following statements cannot use the same shared SQL area:

  ```
  SELECT * FROM emp;
  SELECT *  FROM emp;
  ```

  These statements also cannot use the same shared SQL area:

```
SELECT * FROM emp;
SELECT * FROM Emp;
```

- Statements that differ only in the literals can use the same shared SQL area. For example, the following two statements are considered similar:

  ```
  INSERT INTO T VALUES(1, 'foo', 4)
  INSERT INTO T VALUES(2, 'bar', 7)
  ```

  See Also: Such statements can use the same shared SQL area only when CURSOR_SHARING = FORCE. This is explained more later in this section. For more information on the CURSOR_SHARING parameter, see *Oracle8i SQL Reference*.

- References to schema objects in the SQL statements or PL/SQL blocks must resolve to the same object in the same schema.

  For example, if the schemas of the users Bob and Ed both contain an emp table, and if both users issue the following statement, then their statements cannot use the same shared SQL area:

  ```
  SELECT * FROM emp;
  ```

  If both statements query the same table and qualify the table with the schema, as in the following statement, then they can use the same shared SQL area:

  ```
  SELECT * FROM bob.emp;
  ```

- Bind variables in the SQL statements must match in name and datatype. For example, these statements cannot use the same shared SQL area:

  ```
  SELECT * FROM emp WHERE deptno = :department_no;
  SELECT * FROM emp WHERE deptno = :d_no;
  ```

- The SQL statements must be optimized using the same optimization approach and, in the case of the cost-based approach, the same optimization goal.

  See Also: For information on optimization approach and goal, see Chapter 9, "Optimizing SQL Statements".

Shared SQL areas are most useful for reducing library cache misses for multiple users running the same application. Discuss these criteria with the developers of such applications and agree on strategies to ensure that the SQL statements and PL/SQL blocks of an application can use the same shared SQL areas:

- Use bind variables rather than explicitly specified constants in your statements whenever possible.

  For example, the following two statements cannot use the same shared area because they do not match character for character:

  ```
  SELECT ename, empno FROM emp WHERE deptno = 10;
  SELECT ename, empno FROM emp WHERE deptno = 20;
  ```

  You can accomplish the goals of these statements by using the following statement that contains a bind variable, binding 10 for one occurrence of the statement and 20 for the other:

  ```
  SELECT ename, empno FROM emp WHERE deptno = :department_no;
  ```

  The two occurrences of the statement can then use the same shared SQL area.

- The `CURSOR_SHARING` parameter may solve some performance problems. It has the following values: `FORCE` and `EXACT` (default).

Setting `CURSOR_SHARING` to `FORCE` forces similar statements to share SQL by replacing literals with system generated bind variables. Replacing literals with bind variables improves cursor sharing with reduced memory usage, faster parses, and reduced latch contention.

The `V$SQL_BIND_METADATA` and `V$SQL_BIND_DATA` views show the transformed text. These tables show bind metadata and bind data for all bind variables, including system generated bind variables. System generated bind variables can be distinguished from user bind variables based on the value of `SHARED_FLAG2` in `V$SQL_BIND_DATA`.

For example, the following statement shows bind data only for system generated bind variables.

```
SELECT *
FROM V$SQL_BIND_DATA
WHERE BITAND(SHARED_FLAG2, 256) = 256;
```

This parameter should be set to `FORCE` only when the risk of suboptimal plans is outweighed by the improvements in cursor sharing.

Note: Setting `CURSOR_SHARING` to `FORCE` causes an increase in the maximum lengths (as returned by `DESCRIBE`) of any selected expressions that contain literals (in a `SELECT` statement). However, the actual length of the data returned will not change.

You should consider setting CURSOR_SHARING to FORCE if you can answer 'yes' to both of the following questions:

1. Are there statements in the shared pool that differ only in the values of literals?
2. Is the response time low due to a very high number of library cache misses?

Setting CURSOR_SHARING to EXACT allows SQL statements to share the SQL area only when their texts match exactly.

Note: Oracle does not recommend setting CURSOR_SHARING to FORCE in a DSS environment or if you are using complex queries, query rewrite, or stored outlines.

- Be sure that users of the application do not change the optimization approach and goal for their individual sessions.
- You can also increase the likelihood that SQL statements issued by different applications can share SQL areas by establishing these policies among the developers of the applications:
  - Standardize naming conventions for bind variables and spacing conventions for SQL statements and PL/SQL blocks.
  - Use stored procedures whenever possible. Multiple users issuing the same stored procedure automatically use the same shared PL/SQL area. Because stored procedures are stored in a parsed form, they eliminate run-time parsing altogether.

Use CURSOR_SPACE_FOR_TIME to Speed Access to Shared SQL Areas If you have no library cache misses, then you may still be able to accelerate execution calls by setting the value of the initialization parameter CURSOR_SPACE_FOR_TIME. This parameter specifies whether a shared SQL area can be deallocated from the library cache to make room for a new SQL statement. CURSOR_SPACE_FOR_TIME has the following values meanings:

- If this is set to false (the default), then a shared SQL area can be deallocated from the library cache regardless of whether application cursors associated with its SQL statement are open. In this case, Oracle must verify that a shared SQL area containing the SQL statement is in the library cache.
- If this is set to true, then a shared SQL area can be deallocated only when all application cursors associated with its statement are closed. In this case, Oracle need not verify that a shared SQL area is in the cache, because the shared SQL

area can never be deallocated while an application cursor associated with it is open.

Setting the value of the parameter to `true` saves Oracle a small amount of time and may slightly improve the performance of execution calls. This value also prevents the deallocation of private SQL areas until associated application cursors are closed.

Do not set the value of `CURSOR_SPACE_FOR_TIME` to `true` if you have found library cache misses on execution calls. Such library cache misses indicate that the shared pool is not large enough to hold the shared SQL areas of all concurrently open cursors. If the value is `true`, and if the shared pool has no space for a new SQL statement, then the statement cannot be parsed, and Oracle returns an error saying that there is no more shared memory. If the value is `false`, and if there is no space for a new statement, then Oracle deallocates an existing shared SQL area. Although deallocating a shared SQL area results in a library cache miss later, it is preferable to an error halting your application because a SQL statement cannot be parsed.

Do not set the value of `CURSOR_SPACE_FOR_TIME` to `true` if the amount of memory available to each user for private SQL areas is scarce. This value also prevents the deallocation of private SQL areas associated with open cursors. If the private SQL areas for all concurrently open cursors fills the user's available memory so that there is no space to allocate a private SQL area for a new SQL statement, then the statement cannot be parsed, and Oracle returns an error indicating that there is not enough memory.

Caching Session Cursors If an application repeatedly issues parse calls on the same set of SQL statements, then the reopening of the session cursors can affect system performance. Session cursors can be stored in a session cursor cache. This feature can be particularly useful for applications designed using Oracle Forms, because switching from one form to another closes all session cursors associated with the first form.

Oracle uses the shared SQL area to determine whether more than three parse requests have been issued on a given statement. If so, Oracle assumes the session cursor associated with the statement should be cached and moves the cursor into the session cursor cache. Subsequent requests to parse that SQL statement by the same session then find the cursor in the session cursor cache.

To enable caching of session cursors, you must set the initialization parameter `SESSION_CACHED_CURSORS`. The value of this parameter is a positive integer specifying the maximum number of session cursors kept in the cache. An LRU (Least Recently Used) algorithm removes entries in the session cursor cache to make room for new entries when needed.

You can also enable the session cursor cache dynamically with the statement:

```
ALTER SESSION SET SESSION_CACHED_CURSORS.
```

To determine whether the session cursor cache is sufficiently large for your instance, you can examine the session statistic "session cursor cache hits" in the V$SESSTAT view. This statistic counts the number of times a parse call found a cursor in the session cursor cache. If this statistic is a relatively low percentage of the total parse call count for the session, then you should consider setting SESSION_CACHED_CURSORS to a larger value.

Tuning the Data Dictionary Cache

This section describes how to tune the data dictionary cache with the following:

- Monitoring Data Dictionary Cache Activity
- Reducing Data Dictionary Cache Misses

Monitoring Data Dictionary Cache Activity Determine whether misses on the data dictionary cache are affecting the performance of Oracle. You can examine cache activity by querying the V$ROWCACHE table as described in the following sections.

Misses on the data dictionary cache are to be expected in some cases. Upon instance startup, the data dictionary cache contains no data, so any SQL statement issued is likely to result in cache misses. As more data is read into the cache, the likelihood of cache misses should decrease. Eventually the database should reach a *steady state*, in which the most frequently used dictionary data is in the cache. At this point, very few cache misses should occur. To tune the cache, examine its activity only after your application has been running.

Statistics reflecting data dictionary activity are kept in the dynamic performance table V$ROWCACHE. By default, this table is available only to the user SYS and to users granted SELECT ANY TABLE system privilege, such as SYSTEM.

Each row in this table contains statistics for a single type of the data dictionary item. These statistics reflect all data dictionary activity since the most recent instance startup. These columns in the V$ROWCACHE table reflect the use and effectiveness of the data dictionary cache:

PARAMETER — Identifies a particular data dictionary item. For each row, the value in this column is the item prefixed by dc_. For example, in the row that contains statistics for file descriptions, this column has the value dc_files.

GETS	Shows the total number of requests for information on the corresponding item. For example, in the row that contains statistics for file descriptions, this column has the total number of requests for file descriptions data.
GETMISSES	Shows the number of data requests resulting in cache misses.

Use the following query to monitor the statistics in the V$ROWCACHE table over a period of time while your application is running:

```
SELECT SUM(GETS)  "DATA DICTIONARY GETS",
SUM(GETMISSES) "DATA DICTIONARY CACHE GET MISSES"
FROM V$ROWCACHE;
```

The output of this query could look like this:

```
DATA DICTIONARY GETS  DATA DICTIONARY CACHE GET MISSES
--------------------  --------------------------------
1439044                                 3120
```

Examining the data returned by the sample query leads to these observations:

- The sum of the GETS column indicates that there was a total of 1,439,044 requests for dictionary data.
- The sum of the GETMISSES column indicates that 3120 of the requests for dictionary data resulted in cache misses.
- The ratio of the sums of GETMISSES to GETS is about 0.2%.

Reducing Data Dictionary Cache Misses Examine cache activity by monitoring the sums of the GETS and GETMISSES columns. For frequently accessed dictionary caches, the ratio of total GETMISSES to total GETS should be less than 10% or 15%. If the ratio continues to increase above this threshold while your application is running, then you should consider increasing the amount of memory available to the data dictionary cache. To increase the memory available to the cache, increase the value of the initialization parameter SHARED_POOL_SIZE. The maximum value for this parameter depends on your operating system.

Tuning the Large Pool and Shared Pool for the MTS Architecture

Oracle recommends using the large pool to allocate MTS-related UGA (User Global Area), not the shared pool. This is because Oracle uses the shared pool to allocate SGA (Shared Global Area) memory for other purposes, such as shared SQL and

PL/SQL procedures. Using the large pool, instead of the shared pool, will decrease fragmentation of the shared pool.

To store MTS-related UGA in the large pool, specify a value for the parameter LARGE_POOL_SIZE. LARGE_POOL_SIZE does not have a default value, but its minimal value is 300K. If you do not set a value for LARGE_POOL_SIZE, then Oracle uses the shared pool for MTS user session memory. Oracle has a default value for SHARED_POOL_SIZE of 8MB on 32-bit systems and 64MB on 64 bit systems.

Configure the size of the large pool based on the number of simultaneously active sessions. Each application requires a different amount of memory for session information, and your configuration of the large pool or SGA should reflect the memory requirement. For example, in some applications, MTS requires 200K - 300K to store session information for each active session. If you anticipate 100 active sessions simultaneously, then you should configure the large pool to be 30M, or increase the shared pool accordingly if the large pool is not configured.

Note: If MTS is used, then Oracle allocates some fixed amount of memory (about 10K) per configured session from the shared pool, even if you have configured the large pool. The MTS_CIRCUITS initialization parameter specifies the maximum number of concurrent MTS connections that the database allows. For information on the MTS_CIRCUITS parameter, see *Oracle8i Reference.*

Determining an Effective Setting for MTS UGA Storage The exact amount of UGA Oracle uses depends on each application. To determine an effective setting for the large or shared pools, observe UGA use for a typical user, and multiply this amount by the estimated number of user sessions.

Even though use of shared memory increases with MTS, the total amount of memory use decreases. This is because there are fewer processes, and therefore, Oracle uses less PGA memory with MTS when compared to dedicated server environments.

Note: For best performance with sorts using MTS, set SORT_AREA_SIZE and SORT_AREA_RETAINED_SIZE to the same value. This keeps the sort result in the large pool instead of having it written to disk.

Limiting Memory Use Per User Session by Setting PRIVATE_SGA You can set the PRIVATE_SGA parameter to limit the memory used by each client session from the SGA. PRIVATE_SGA defines the number of bytes of memory used from the SGA by a session. However, this parameter is rarely used because most DBAs do not limit SGA consumption an a user-by-user basis.

> **See Also:** For more information, see the *Oracle8i Reference.*

Reducing Memory Use With Three-Tier Connections If you have a high number of connected users, then you can reduce memory use to an acceptable level by implementing "three-tier connections". This by-product of using a TP monitor is feasible only with pure transactional models, because locks and uncommitted DMLs cannot be held between calls. MTS is much less restrictive of the application design than a TP monitor. It dramatically reduces operating system process count and context switches by enabling users to share a pool of servers. MTS also substantially reduces overall memory usage even though more SGA is used in MTS mode.

> **Note:** On NT, shared servers are implemented as *threads* instead of processes.

The V$SESSTAT View Oracle collects statistics on total memory used by a session and stores them in the dynamic performance view V$SESSTAT. By default, this view is available only to the user SYS and to users granted SELECT ANY TABLE system privilege, such as SYSTEM. These statistics are useful for measuring session memory use:

session UGA memory	The value of this statistic is the amount of memory in bytes allocated to the session.
session UGA memory max	The value of this statistic is the maximum amount of memory in bytes ever allocated to the session.

To find the value, query V$STATNAME as described in "Technique 3" on page 19-9.

You can use the following query to decide how much larger to make the shared pool if you are using a Multi-threaded Server. Issue these queries while your application is running:

```
SELECT SUM(VALUE) || ' BYTES' "TOTAL MEMORY FOR ALL SESSIONS"
    FROM V$SESSTAT, V$STATNAME
    WHERE NAME = 'SESSION UGA MEMORY'
    AND V$SESSTAT.STATISTIC# = V$STATNAME.STATISTIC#;

SELECT SUM(VALUE) || ' BYTES' "TOTAL MAX MEM FOR ALL SESSIONS"
    FROM V$SESSTAT, V$STATNAME
    WHERE NAME = 'SESSION UGA MEMORY MAX'
    AND V$SESSTAT.STATISTIC# = V$STATNAME.STATISTIC#;
```

These queries also select from the dynamic performance table `V$STATNAME` to obtain internal identifiers for *session memory* and *max session memory*. The results of these queries could look like this:

```
TOTAL MEMORY FOR ALL SESSIONS
-----------------------------
157125 BYTES

TOTAL MAX MEM FOR ALL SESSIONS
------------------------------
417381 BYTES
```

The result of the first query indicates that the memory currently allocated to all sessions is 157,125 bytes. This value is the total memory whose location depends on how the sessions are connected to Oracle. If the sessions are connected to dedicated server processes, then this memory is part of the memories of the user processes. If the sessions are connected to shared server processes, then this memory is part of the shared pool.

The result of the second query indicates that the sum of the maximum sizes of the memories for all sessions is 417,381 bytes. The second result is greater than the first, because some sessions have deallocated memory since allocating their maximum amounts.

You can use the result of either of these queries to determine how much larger to make the shared pool if you use a Multi-threaded Server. The first value is likely to be a better estimate than the second, unless nearly all sessions are likely to reach their maximum allocations at the same time.

Tuning Reserved Space from the Shared Pool

On busy systems, the database may have difficulty finding a contiguous piece of memory to satisfy a large request for memory. This search may disrupt the behavior of the shared pool, leading to fragmentation and poor performance.

You can reserve memory within the shared pool to satisfy large allocations during operations such as PL/SQL compilation and trigger compilation. Smaller objects do not fragment the reserved list, helping to ensure that the reserved list has large contiguous chunks of memory. After the memory allocated from the reserved list is freed, it returns to the reserved list.

Controlling Space Reclamation of the Shared Pool The ABORTED_REQUEST_THRESHOLD procedure in the DBMS_SHARED_POOL package lets you limit the size of allocations allowed to flush the shared pool if the free lists cannot satisfy the request size. The database incrementally flushes unused objects from the shared pool until there is sufficient memory to satisfy the allocation request. In most cases, this frees enough memory for the allocation to complete successfully.

If the database flushes all objects currently not in use on the system without finding a large enough piece of contiguous memory, then an error occurs. Flushing all objects, however, affects other users on the system, as well as system performance. The ABORTED_REQUEST_THRESHOLD procedure lets you localize the error to the process that could not allocate memory.

Using SHARED_POOL_RESERVED_SIZE The size of the reserved list, and the minimum size of the objects that can be allocated from the reserved list, can be controlled by the initialization parameter SHARED_POOL_RESERVED_SIZE. Begin this tuning only after performing all other shared pool tuning.

The default value for SHARED_POOL_RESERVED_SIZE is 5% of the SHARED_POOL_SIZE. This means that, by default, the reserved list is always configured.

If SHARED_POOL_RESERVED_SIZE > 1/2 SHARED_POOL_SIZE, then Oracle signals an error. Ideally, this parameter should be large enough to satisfy any request scanning for memory on the reserved list without flushing objects from the shared pool. The amount of operating system memory, however, may constrain the size of the shared pool. In general, set SHARED_POOL_RESERVED_SIZE to 10% of SHARED_POOL_SIZE. For most systems, this value is sufficient if you have already tuned the shared pool. If you increase this value, then the database allows fewer allocations from the reserved list and requests more memory from the shared pool list.

Statistics from the V$SHARED_POOL_RESERVED view help you tune these parameters. On a system with ample free memory to increase the size of the SGA, the goal is to have REQUEST_MISSES = 0. If the system is constrained for operating system memory, then the goal is to not have REQUEST_FAILURES or at least prevent this value from increasing.

If you cannot achieve this, then increase the value for SHARED_POOL_RESERVED_SIZE. Also, increase the value for SHARED_POOL_SIZE by the same amount, because the reserved list is taken from the shared pool.

> **See Also:** For details on setting the LARGE_POOL_SIZE parameter, see *Oracle8i Reference*.

SHARED_POOL_ RESERVED_SIZE Too Small The reserved pool is too small when the value for REQUEST_FAILURES is more than zero and increasing. To resolve this, increase the value for the SHARED_POOL_RESERVED_SIZE and SHARED_POOL_SIZE accordingly. The settings you select for these depend on your system's SGA size constraints.

This option increases the amount of memory available on the reserved list without having an effect on users who do not allocate memory from the reserved list. As a second option, reduce the number of allocations allowed to use memory from the reserved list; however, doing so increases the normal shared pool, which may have an effect on other users on the system.

SHARED_POOL_ RESERVED_SIZE Too Large Too much memory may have been allocated to the reserved list if:

- REQUEST_MISS = 0 or not increasing
- FREE_MEMORY = > 50% of SHARED_POOL_RESERVED_SIZE minimum

If either of these is true, then decrease the value for SHARED_POOL_RESERVED_SIZE.

SHARED_POOL_SIZE Too Small The V$SHARED_POOL_RESERVED fixed table can also indicate when the value for SHARED_POOL_SIZE is too small. This may be the case if REQUEST_FAILURES > 0 and increasing.

If you have enabled the reserved list, then decrease the value for SHARED_POOL_RESERVED_SIZE. If you have not enabled the reserved list, then you could increase SHARED_POOL_SIZE.

Tuning the Buffer Cache

You can use or bypass the Oracle buffer cache for particular operations. Oracle bypasses the buffer cache for sorting and parallel reads. For operations that use the buffer cache, this section explains:

- Evaluating Buffer Cache Activity with the Cache Hit Ratio

- Increasing the Cache Hit Ratio by Reducing Buffer Cache Misses
- Removing Unnecessary Buffers when Cache Hit Ratio Is High
- Accommodating LOBs in the Buffer Cache

After tuning private SQL and PL/SQL areas and the shared pool, you can devote the remaining available memory to the buffer cache. It may be necessary to repeat the steps of memory allocation after the initial pass through the process. Subsequent passes allow you to make adjustments in earlier steps based on changes in later steps. For example, if you increase the size of the buffer cache, then you may need to allocate more memory to Oracle to avoid paging and swapping.

Evaluating Buffer Cache Activity with the Cache Hit Ratio

Physical I/O takes a significant amount of time, typically in excess of 15 milliseconds. Physical I/O also increases the CPU resources required, owing to the path length in device drivers and operating system event schedulers. Your goal is to reduce this overhead as much as possible by making it more likely that the required block is in memory. The extent to which you achieve this is measured using the cache hit ratio. Within Oracle, this term applies specifically to the database buffer cache.

Calculating the Cache Hit Ratio Oracle collects statistics that reflect data access and stores them in the dynamic performance view `V$SYSSTAT`. By default, this table is available only to the user `SYS` and to users, such as `SYSTEM`, who have the `SELECT ANY TABLE` system privilege. Information in the `V$SYSSTAT` view can also be obtained through the Simple Network Management Protocol (SNMP).

These statistics are useful for tuning the buffer cache:

`DB BLOCK GETS`, `CONSISTENT GETS`	The sum of these values is the total number of requests for data. This value includes requests satisfied by access to buffers in memory.
`PHYSICAL READS`	This statistic is the total number of requests for data resulting in access to datafiles on disk.

Monitor these statistics as follows over a period of time while your application is running:

```
SELECT NAME, VALUE
FROM V$SYSSTAT
WHERE NAME IN ('DB BLOCK GETS', 'CONSISTENT GETS', 'PHYSICAL READS');
```

The output of this query could look like the following:

```
NAME                                                       VALUE
------------------------------------------------------ ----------
DB BLOCK GETS                                              85792
CONSISTENT GETS                                           278888
PHYSICAL READS                                             23182
```

Calculate the hit ratio for the buffer cache with this formula:

Hit Ratio = 1 - (physical reads / (db block gets + consistent gets))

Based on the statistics obtained by the example query, the buffer cache hit ratio is 94%.

Buffer Pinning Statistics These statistics are useful in evaluating buffer pinning:

Buffer pinned	This statistic measures the number of times a buffer was already pinned by a client when a client checks to determine if the buffer it wants is already pinned.
Buffer not pinned	This statistic measures the number of times the buffer was not pinned by the client when a client checks to determine if the buffer it wants is already pinned.

These statistics are not incremented when a client performs such a check before releasing it, because the client does not intend to use the buffer in this case.

These statistics provide a measure of how often a long consistent read pin on a buffer is beneficial. If the client is able to reuse the pinned buffer many times, then it indicates that it is useful to have the buffer pinned.

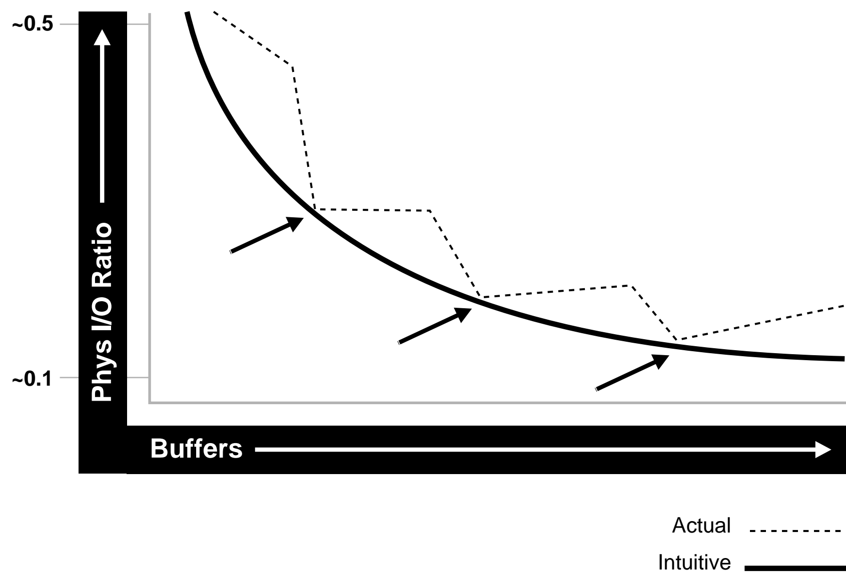
Evaluating the Cache Hit Ratio When looking at the cache hit ratio, remember that blocks encountered during a "long" full table scan are not put at the head of the LRU list; therefore, repeated scanning does not cause the blocks to be cached.

Repeated scanning of the same large table is rarely the most efficient approach. It may be better to perform all of the processing in a single pass, even if this means that the overnight batch suite can no longer be implemented as a SQL*Plus script that contains no PL/SQL. The solution lies at the design or implementation level.

Production sites running with thousands, or tens of thousands, of buffers rarely use memory effectively. In any large database running OLTP applications, in any given unit of time, most rows are accessed either one or zero times. On this basis, there is little point in keeping the row, or the block that contains it, in memory for very long following its use.

Finally, the relationship between the cache hit ratio and the number of buffers is far from a smooth distribution. When tuning the buffer pool, avoid the use of additional buffers that contribute little or nothing to the cache hit ratio. As illustrated in Figure 19–2, only narrow bands of values of DB_BLOCK_BUFFERS are worth considering.

Figure 19–2 Buffer Pool Cache Hit Ratio



> **Note:** A common mistake is to continue increasing the value of DB_BLOCK_BUFFERS. Such increases have no effect if you are doing full table scans or other operations that do not use the buffer cache.

As a general rule, increase DB_BLOCK_BUFFERS when:

- The cache hit ratio is less than 0.9.
- There is no evidence of undue page faulting.
- The previous increase of DB_BLOCK_BUFFERS was effective.

Determining which Buffers are in the Pool The CATPARR.SQL script creates the view V$BH, which shows the file number and block number of blocks that currently reside within the SGA. Although CATPARR.SQL is primarily intended for use in parallel server environments, you can run it as SYS even if you're operating a single instance environment.

Perform a query similar to the following:

```
SELECT file#, COUNT(block#), COUNT (DISTINCT file# || block#)
FROM V$BH
GROUP BY file#;
```

Increasing the Cache Hit Ratio by Reducing Buffer Cache Misses

If your hit ratio is low, or less than 60% or 70%, then you may want to increase the number of buffers in the cache to improve performance. To make the buffer cache larger, increase the value of the initialization parameter DB_BLOCK_BUFFERS.

Removing Unnecessary Buffers when Cache Hit Ratio Is High

If your hit ratio is high, then your cache is probably large enough to hold your most frequently accessed data. In this case, you may be able to reduce the cache size and still maintain good performance. To make the buffer cache smaller, reduce the value of the initialization parameter DB_BLOCK_BUFFERS. The minimum value for this parameter is 50, but it is unusual for systems to run with less than 1,000. You can use any leftover memory for other Oracle memory structures, for example, if you're using parallel query.

Accommodating LOBs in the Buffer Cache

Both temporary and permanent LOBs can use the buffer cache.

Temporary LOBs created with the CACHE parameter set to TRUE move through the buffer cache. Temporary LOBs created with the CACHE parameter set to FALSE are read directly from, and written directly to, disk.

You can use durations for automatic cleanup to save time and effort. Also, it is more efficient for the database to end a duration and free all temporary LOBs associated with a duration than it is to free each one explicitly.

Temporary LOBs create entirely new copies of themselves on assignments. For example:

```
LOCATOR1 BLOB;
LOCATOR2 BLOB;
DBMS_LOB.CREATETEMPORARY (LOCATOR1,TRUE,DBMS_LOB.SESSION);
   LOCATOR2 := LOCATOR;
```

The above code causes a copy of the temporary LOB pointed to by LOCATOR1 to be created. You may also want to consider using pass by reference semantics in PL/SQL.

Or, in OCI, you may declare pointers to locators as in the following example:

```
OCILOBDESCRIPTOR *LOC1;
OCILOBDESCRIPTOR *LOC2;
OCILOBCREATETEMPORARY (LOC1,TRUE,OCIDURATIONSESSION);
LOC2 = LOC1;
```

Avoid using OCILobAssign() statements, because these also cause deep copies of temporary LOBs. In other words, a new copy of the temporary LOB is created.

Pointer assignment does not cause deep copies; it just causes pointers to point to the same thing.

Tuning Multiple Buffer Pools

This section covers:

- Overview of the Multiple Buffer Pool Feature
- When to Use Multiple Buffer Pools
- Tuning the Buffer Cache Using Multiple Buffer Pools
- Enabling Multiple Buffer Pools
- Using Multiple Buffer Pools
- Dictionary Views Showing Default Buffer Pools
- Sizing Each Buffer Pool
- Identifying and Eliminating LRU Latch Contention

Overview of the Multiple Buffer Pool Feature

Schema objects are referenced with varying usage patterns; therefore, their cache behavior may be quite different. Multiple buffer pools enable you to address these differences. You can use a KEEP buffer pool to maintain objects in the buffer cache and a RECYCLE buffer pool to prevent objects from consuming unnecessary space

in the cache. When an object is allocated to a cache, all blocks from that object are placed in that cache. Oracle maintains a DEFAULT buffer pool for objects that have not been assigned to one of the buffer pools.

Each buffer pool in Oracle comprises a number of working sets. A different number of sets can be allocated for each buffer pool. All sets use the same LRU (Least Recently Used) replacement policy. A strict LRU aging policy provides good hit rates in most cases, but you can sometimes improve hit rates by providing some hints.

The main problem with the LRU list occurs when a very large segment is accessed frequently in a random fashion. Here, *very large* means large compared to the size of the cache. Any single segment that accounts for a substantial portion (more than 10%) of nonsequential physical reads is probably one of these segments. Random reads to such a large segment can cause buffers that contain data for other segments to be aged out of the cache. The large segment ends up consuming a large percentage of the cache, but does not benefit from the cache.

Very frequently accessed segments are not affected by large segment reads, because their buffers are warmed frequently enough that they do not age out of the cache. The main trouble occurs with "warm" segments that are not accessed frequently enough to survive the buffer flushing caused by the large segment reads.

You have two options for solving this problem. One option is to move the large segment into a separate RECYCLE cache, so that it does not disturb the other segments. The RECYCLE cache should be smaller than the DEFAULT buffer pool, and it should reuse buffers more quickly than the DEFAULT buffer pool.

The other option is to move the small warm segments into a separate KEEP cache that is not used at all for large segments. The KEEP cache can be sized to minimize misses in the cache. You can make the response times for specific queries more predictable by putting the segments accessed by the queries in the KEEP cache to ensure that they are never aged out.

When to Use Multiple Buffer Pools

When you examine system I/O performance, you should analyze the schema and determine whether multiple buffer pools would be advantageous. Consider a KEEP cache if there are small, frequently accessed tables that require quick response time. Very large tables with random I/O are good candidates for a RECYCLE cache.

Use the following steps to determine the percentage of the cache used by an individual object at a given point in time:

1. Find the Oracle internal object number of the segment by entering the following:

   ```
   SELECT DATA_OBJECT_ID, OBJECT_TYPE
   FROM USER_OBJECTS
   WHERE OBJECT_NAME = '<SEGMENT_NAME>';
   ```

   Because two objects can have the same name (if they are different types of objects), you can use the OBJECT_TYPE column to identify the object of interest. If the object is owned by another user, then use the view DBA_OBJECTS or ALL_OBJECTS instead of USER_OBJECTS.

2. Find the number of buffers in the buffer cache for SEGMENT_NAME:

   ```
   SELECT COUNT(*) BUFFERS
   FROM V$BH
   WHERE OBJD = <DATA_OBJECT_ID>;
   ```

   where DATA_OBJECT_ID is from Step 1.

3. Find the total number of buffers in the instance:

   ```
   SELECT VALUE "TOTAL BUFFERS"
   FROM  V$PARAMETER
   WHERE NAME = 'DB_BLOCK_BUFFERS';
   ```

4. Calculate the ratio of buffers to total buffers, to obtain the percentage of the cache currently used by SEGMENT_NAME.

$$\%\ \text{cache used by}\ \textit{segment_name} = \frac{\textit{buffers}\quad \text{(Step 2)}}{\textit{total buffers}\quad \text{(Step 3)}}$$

> **Note:** This technique works only for a single segment. You must run the query for each partition for a partitioned object.

If the number of local block gets equals the number of physical reads for statements involving such objects, then consider using a RECYCLE cache, because of the limited usefulness of the buffer cache for the objects.

Tuning the Buffer Cache Using Multiple Buffer Pools

When you partition your buffer cache into multiple buffer pools, each buffer pool can be used for blocks from objects that are accessed in different ways. If the blocks of a particular object are likely to be reused, then you should pin that object in the buffer cache so that the next use of the block does not require disk I/O. Conversely, if a block probably will not be reused within a reasonable period of time, then discard it to make room for more frequently used blocks.

By properly allocating objects to appropriate buffer pools, you can:

- Reduce or eliminate I/Os.
- Isolate an object in the cache.
- Restrict or limit an object to a part of the cache.

Enabling Multiple Buffer Pools

You can create multiple buffer pools for each database instance. The same set of buffer pools need not be defined for each instance of the database. Among instances, the buffer pools may be different sizes or not defined at all. Tune each instance separately.

Defining New Buffer Pools You can define each buffer pool using the `BUFFER_POOL_` *`name`* initialization parameter. You can specify two attributes for each buffer pool: the number of buffers in the buffer pool, and the number of LRU latches allocated to the buffer pool.

The initialization parameters used to define buffer pools are:

`BUFFER_POOL_KEEP`	Defines the `KEEP` buffer pool.
`BUFFER_POOL_RECYCLE`	Defines the `RECYCLE` buffer pool.
`DB_BLOCK_BUFFERS`	Defines the number of buffers for the database instance. Each individual buffer pool is created from this total amount with the remainder allocated to the `DEFAULT` buffer pool.
`DB_BLOCK_LRU_LATCHES`	Defines the number of LRU latches for the entire database instance. Each buffer pool defined takes from this total in a fashion similar to `DB_BLOCK_BUFFERS`.

For example:

```
BUFFER_POOL_KEEP = #buffers
                  |(buffers:#buffers, lru_latches:#latches)
                  |(lru_latches:#latches, buffers:#buffers)
                  |(buffers:#buffers)
```

The size of each buffer pool is subtracted from the total number of buffers defined for the entire buffer cache (that is, the value of the DB_BLOCK_BUFFERS parameter). The aggregate number of buffers in all buffer pools cannot, therefore, exceed this value. Likewise, the number of LRU latches allocated to each buffer pool is taken from the total number allocated to the instance by the DB_BLOCK_LRU_LATCHES parameter. If either constraint is violated, then Oracle displays an error, and the database is not mounted.

The minimum number of buffers you must allocate to each buffer pool is 50 times the number of LRU latches. For example, a buffer pool with 3 LRU latches must have at least 150 buffers.

Oracle automatically defines three buffer pools: KEEP, RECYCLE, and DEFAULT. The DEFAULT buffer pool always exists. You do not explicitly define the size of the DEFAULT buffer pool or the number of working sets assigned to the DEFAULT buffer pool. Rather, each value is inferred from the total number allocated minus the number allocated to every other buffer pool. There is no requirement that any one buffer pool be defined for another buffer pool to be used.

Using Multiple Buffer Pools

This section describes how to establish a DEFAULT buffer pool for an object. All blocks for the object go in the specified buffer pool.

The BUFFER_POOL clause is used to define the DEFAULT buffer pool for an object. This clause is valid for CREATE and ALTER table, cluster, and index DDL statements. The buffer pool name is case insensitive. The blocks from an object without an explicitly set buffer pool go into the DEFAULT buffer pool.

If a buffer pool is defined for a partitioned table or index, then each partition of the object inherits the buffer pool from the table or index definition unless you override it with a specific buffer pool.

When the DEFAULT buffer pool of an object is changed using the ALTER statement, all buffers currently containing blocks of the altered segment remain in the buffer pool they were in before the ALTER statement. Newly loaded blocks and any blocks that have aged out and are reloaded go into the new buffer pool.

The syntax of the BUFFER_POOL clause is: BUFFER_POOL {KEEP | RECYCLE | DEFAULT}

For example:

```
BUFFER_POOL KEEP
```

or

```
BUFFER_POOL RECYCLE
```

The following DDL statements accept the buffer pool clause:

- `CREATE TABLE` *table name...* `STORAGE` *(buffer_pool_clause)*

  A buffer pool is not permitted for a clustered table. The buffer pool for a clustered table is specified at the cluster level.

  For an index-organized table, a buffer pool can be defined on both the index and the overflow segment.

  For a partitioned table, a buffer pool can be defined on each partition. The buffer pool is specified as a part of the storage clause for each partition.

  For example:

  ```
  CREATE TABLE table_name (col_1 NUMBER, col_2 NUMBER)
  PARTITION BY RANGE (col_1)
  (PARTITION ONE VALUES LESS THAN (10)
  STORAGE (INITIAL 10K BUFFER_POOL RECYCLE),
  PARTITION TWO VALUES LESS THAN (20) STORAGE (BUFFER_POOL KEEP));
  ```

- `CREATE INDEX` *index name...* `STORAGE` *(buffer_pool_clause)*

  For a global or local partitioned index, a buffer pool can be defined on each partition.

- `CREATE CLUSTER` *cluster_name...*`STORAGE` (*buffer_pool_clause)*
- `ALTER TABLE` *table_name...* `STORAGE` (*buffer_pool_clause)*

  A buffer pool can be defined during simple `ALTER TABLE`, `MODIFY PARTITION`, `MOVE PARTITION`, `ADD PARTITION`, and `SPLIT PARTITION` statements for both new partitions.

- `ALTER INDEX` *index_name...* `STORAGE` (*buffer_pool_clause)*

  A buffer pool can be defined during simple `ALTER INDEX`, `REBUILD`, `MODIFY PARTITION`, `SPLIT PARTITION` statements for both new partitions, and rebuild partitions.

- `ALTER CLUSTER` *cluster_name...* `STORAGE` (*buffer_pool_clause)*

Dictionary Views Showing Default Buffer Pools

The following dictionary views have a BUFFER POOL column indicating the DEFAULT buffer pool for the given object.

USER_CLUSTERS	ALL_CLUSTERS	DBA_CLUSTERS
USER_INDEXES	ALL_INDEXES	DBA_INDEXES
USER_SEGMENTS	DBA_SEGMENTS	
USER_TABLES	USER_OBJECT_TABLES	USER_ALL_TABLES
ALL_TABLES	ALL_OBJECT_TABLES	ALL_ALL_TABLES
DBA_TABLES	DBA_OBJECT_TABLES	DBA_ALL_TABLES
USER_PART_TABLES	ALL_PART_TABLES	DBA_PART_TABLES
USER_PART_INDEXES	ALL_PART_INDEXES	DBA_PART_INDEXES
USER_TAB_PARTITIONS	ALL_TAB_PARTITIONS	DBA_TAB_PARTITIONS
USER_IND_PARTITIONS	ALL_IND_PARTITIONS	DBA_IND_PARTITIONS

The views V$BUFFER_POOL_STATISTICS and GV$BUFFER_POOL_STATISTICS describe the buffer pools allocated on the local instance and entire database, respectively. To create these views you must run the CATPERF.SQL file.

Sizing Each Buffer Pool

This section explains how to size the following:

- KEEP Buffer Pool
- RECYCLE Buffer Pool

KEEP Buffer Pool The goal of the KEEP buffer pool is to retain objects in memory, thus avoiding I/O operations. The size of the KEEP buffer pool, therefore, depends on the objects that you want to keep in the buffer cache. You can compute an approximate size for the KEEP buffer pool by adding together the sizes of all objects dedicated to this pool. Use the ANALYZE statement to obtain the size of each object. Although the ESTIMATE clause provides a rough measurement of sizes, the COMPUTE STATISTICS clause is preferable because it provides the most accurate value possible.

The buffer pool hit ratio can be determined using the formula:

$$hit\ ratio = 1 - \frac{physical\ reads}{(block\ gets + consistent\ gets)}$$

Where the values of physical reads, block gets, and consistent gets can be obtained for the KEEP buffer pool from the following query:

```
SELECT PHYSICAL_READS, BLOCK_GETS, CONSISTENT_GETS
FROM V$BUFFER_POOL_STATISTICS WHERE NAME = 'KEEP';
```

Note: You must first run the CATPERF.SQL script.

The KEEP buffer pool has a 100% hit ratio only after the buffers have been loaded into the buffer pool. Therefore, do not compute the hit ratio until after the system runs for a while and achieves steady-state performance. Calculate the hit ratio by taking two snapshots of system performance at different times using the above query. Subtract the newest values from the older values for physical reads, block gets, and consistent gets, and use these values to compute the hit ratio.

A 100% buffer pool hit ratio may not be optimal. Often, you can decrease the size of your KEEP buffer pool and still maintain a sufficiently high hit ratio. Allocate blocks removed from use for the KEEP buffer pool to other buffer pools.

Note: If an object grows in size, then it may no longer fit in the KEEP buffer pool. You will begin to lose blocks out of the cache.

Each object kept in memory results in a trade-off: it is beneficial to keep frequently accessed blocks in the cache, but retaining infrequently used blocks results in less space for other, more active blocks.

RECYCLE Buffer Pool The goal of the RECYCLE buffer pool is to eliminate blocks from memory as soon as they are no longer needed. If an application accesses the blocks of a very large object in a random fashion, then there is little chance of reusing a block stored in the buffer pool before it is aged out. This is true regardless of the size of the buffer pool (given the constraint of the amount of available physical memory). Because of this, the object's blocks should not be cached; those cache buffers can be allocated to other objects.

Be careful, however, not to discard blocks from memory too quickly. If the buffer pool is too small, then blocks may age out of the cache before the transaction or SQL

statement has completed execution. For example, an application may select a value from a table, use the value to process some data, and then update the record. If the block is removed from the cache after the select statement, then it must be read from disk again to perform the update. The block should be retained for the duration of the user transaction.

By executing statements with a SQL statement tuning tool, such as Oracle Trace, or with the SQL trace facility enabled and running TKPROF on the trace files, you can get a listing of the total number of data blocks physically read from disk. (This number appears in the "disk" column in the TKPROF output.) The number of disk reads for a particular SQL statement should not exceed the number of disk reads of the same SQL statement with all objects allocated from the DEFAULT buffer pool.

Two other statistics can tell you whether the RECYCLE buffer pool is too small. If the "free buffer waits" statistic ever becomes excessive, then the pool is probably too small. Likewise, the number of "log file sync" wait events will increase. One way to size the RECYCLE buffer pool is to run the system with the RECYCLE buffer pool disabled. At steady state, the number of buffers in the DEFAULT buffer pool being consumed by segments that would normally go in the RECYCLE buffer pool can be divided by four. Use the result as a value for sizing the RECYCLE cache.

Identifying Segments to Put into the KEEP and RECYCLE Buffer Pools A good candidate for a segment to put into the RECYCLE buffer pool is a segment that is at least twice the size of the DEFAULT buffer pool and has incurred at least a few percent of the total I/Os in the system.

A good candidate for a segment to put into the KEEP pool is a segment that is smaller than 10% of the size of the DEFAULT buffer pool and has incurred at least 1% of the total I/Os in the system.

The trouble with these rules is that it can sometimes be difficult to determine the number of I/Os per segment if a tablespace has more than one segment. One way to solve this problem is to sample the I/Os that occur over a period of time by selecting from V$SESSION_WAIT to determine a statistical distribution of I/Os per segment.

Identifying and Eliminating LRU Latch Contention

LRU latches regulate the least recently used buffer lists used by the buffer cache. If there is latch contention, then processes are waiting and spinning before obtaining the latch.

You can set the overall number of latches in the database instance using the DB_BLOCK_LRU_LATCHES parameter. When each buffer pool is defined, a number of

these LRU latches can be reserved for the buffer pool. The buffers of a buffer pool are divided evenly between the LRU latches of the buffer pool.

To determine whether your system is experiencing latch contention, begin by determining whether there is LRU latch contention for any individual latch.

```
SELECT CHILD#, SLEEPS / GETS RATIO
FROM V$LATCH_CHILDREN
WHERE NAME = 'cache buffers lru chain';
```

The miss ratio for each LRU latch should be less than 3%. A ratio above 3% for any particular latch is indicative of LRU latch contention and should be addressed. You can determine the buffer pool to which the latch is associated as follows:

```
SELECT NAME
FROM V$BUFFER_POOL
WHERE lo_setid <= child_latch_number
AND hi_setid >= child_latch_number;
```

Where *child_latch_number* is the *child#* from the previous query.

You can alleviate LRU latch contention by increasing the overall number of latches in the system and the number of latches allocated to the buffer pool indicated in the second query.

The maximum number of latches allowed is the lower of:

number_of_cpus * 2 * 3   or   *number_of_buffers / 50*

This limitation exists because no set can have fewer than 50 buffers. If you specify a value larger than the maximum, then Oracle automatically resets the number of latches to the largest value allowed by the formula.

For example, if the number of CPUs is 4 and the number of buffers is 200, then a maximum of 4 latches would be allowed (minimum of 4*2*3, 200/50). If the number of CPUs is 4 and the number of buffers is 10000, then the maximum number of latches allowed is 24 (minimum of 4*2*3, 10000/50).

Tuning Sort Areas

If large sorts occur frequently, then consider increasing the value of the parameter `SORT_AREA_SIZE` with either or both of two goals in mind:

- Increase the number of sorts that can be conducted entirely within memory.
- Speed up those sorts that cannot be conducted entirely within memory.

Large sort areas can be used effectively if you combine a large SORT_AREA_SIZE with a minimal SORT_AREA_RETAINED_SIZE. If memory is not released until the user disconnects from the database, then large sort work areas could cause problems. The SORT_AREA_RETAINED_SIZE parameter lets you specify the level down to which memory should be released as soon as possible following the sort. Set this parameter to zero if large sort areas are being used in a system with many simultaneous users.

SORT_AREA_RETAINED_SIZE is maintained for each sort operation in a query. Thus, if 4 tables are being sorted for a sort merge, then Oracle maintains 4 areas of SORT_AREA_RETAINED_SIZE.

> **See Also:** For more information, see the "Tuning Sorts" section in Chapter 20, "Tuning I/O".

Reallocating Memory

After resizing your Oracle memory structures, re-evaluate the performance of the library cache, the data dictionary cache, and the buffer cache. If you have reduced the memory consumption of any of these structures, then you may want to allocate more memory to another. For example, if you have reduced the size of your buffer cache, then you may want to use the additional memory by for the library cache.

Tune your operating system again. Resizing Oracle memory structures may have changed Oracle memory requirements. In particular, be sure paging and swapping are not excessive. For example, if the size of the data dictionary cache or the buffer cache has increased, then the SGA may be too large to fit into main memory. In this case, the SGA could be paged or swapped.

While reallocating memory, you may determine that the optimum size of Oracle memory structures requires more memory than your operating system can provide. In this case, you may improve performance even further by adding more memory to your computer.

Reducing Total Memory Usage

If the overriding performance problem is that the server simply does not have enough memory to run the application as currently configured, and the application is logically a single application (that is, it cannot readily be segmented or distributed across multiple servers), then only two possible solutions exist:

- Increase the amount of memory available.
- Decrease the amount of memory used.

The most dramatic reductions in server memory usage always come from reducing the number of database connections, which in turn can resolve issues relating to the number of open network sockets and the number of operating system processes. However, to reduce the number of connections without reducing the number of users, the connections that remain must be shared. This forces the user processes to adhere to a paradigm in which every message request sent to the database describes a complete or *atomic* transaction.

Writing applications to conform to this model is not necessarily either restrictive or difficult, but it is certainly different. Conversion of an existing application, such as an Oracle Forms suite, to conform is not normally possible without a complete rewrite.

The Oracle Multi-threaded Server architecture is an effective solution for reducing the number of server operating system processes. MTS is also quite effective at reducing overall memory requirements. You can also use MTS to reduce the number of network connections when you use MTS with connection pooling and connection concentration.

Shared connections are possible in Oracle Forms environments when you use an intermediate server that is also a client. In this configuration, use the `DBMS_PIPE` package to transmit atomic requests from the user’s individual connection on the intermediate server to a shared daemon in the intermediate server. The daemon, in turn, owns a connection to the central server.

20

Tuning I/O

This chapter explains how to avoid input/output (I/O) bottlenecks that could prevent Oracle from performing at its maximum potential.

This chapter contains the following sections:

- Understanding I/O Problems
- Detecting I/O Problems
- Solving I/O Problems

Understanding I/O Problems

The performance of many software applications is inherently limited by disk input/output (I/O). Often, CPU activity must be suspended while I/O activity completes. Such an application is said to be *I/O bound*. Oracle is designed so that performance is not limited by I/O.

Tuning I/O can enhance performance if a disk containing database files is operating at its capacity. However, tuning I/O cannot help performance in *CPU bound* cases—or cases in which your computer's CPUs are operating at their capacity.

> **See Also:** It is important to tune I/O after following the recommendations presented in Chapter 19, "Tuning Memory Allocation". That chapter explains how to allocate memory so as to reduce I/O to a minimum. After reaching this minimum, follow the instructions in this chapter to achieve more efficient I/O performance.

This section introduces I/O performance issues. It covers:

- Tuning I/O: Top Down and Bottom Up
- Analyzing I/O Requirements
- Planning File Storage
- Choosing Data Block Size
- Evaluating Device Bandwidth

Tuning I/O: Top Down and Bottom Up

When designing a new system, you should analyze I/O needs from the top down, determining what resources you require in order to achieve the desired performance.

For an existing system, you should approach I/O tuning from the bottom up:

1. Determine the number of disks on the system.
2. Determine the number of disks that are being used by Oracle.
3. Determine the type of I/O that your system performs.
4. Ascertain whether the I/Os are going to the file system or to raw devices.

5. Determine how to spread objects over multiple disks, using either manual striping or striping software.
6. Calculate the level of performance you can expect.

Analyzing I/O Requirements

This section explains how to determine your system's I/O requirements.

1. Calculate the total throughput your application requires.

   To begin, figure out the number of reads and writes involved in each transaction, and distinguish the objects against which each operation is performed.

   In an OLTP application, for example, each transaction might involve:

   - 1 read from object A.
   - 1 read from object B.
   - 1 write to object C.

   So, one transaction requires 2 reads and 1 write, all to different objects.

2. Define the I/O performance target for this application by specifying the number of transactions per second (tps) that the system must support.

   With this example, the designer might specify that 100 tps constitutes an acceptable level of performance. To achieve this, the system must be able to perform 300 I/Os per second:

   - 100 reads from object A.
   - 100 reads from object B.
   - 100 writes to object C.

3. Determine the number of disks needed to achieve this level of performance.

   To do this, ascertain the number of I/Os that each disk can perform per second. This number depends on three factors:

   - The speed of your particular disk hardware.
   - Whether the I/Os needed are reads or writes.
   - Whether you are using the file system or raw devices.

   In general, disk speed tends to have the following characteristics:

Table 20–1 Relative Disk Speed

Disk Speed	File System	Raw Devices
Reads per second	fast	slow
Writes per second	slow	fast

4. Write the relative speed per operation of your disks in a chart like the one shown in Table 20–2:

Table 20–2 Disk I/O Analysis Worksheet

Disk Speed	File System	Raw Devices
Reads per second		
Writes per second		

The disks in the current example have characteristics as shown in Table 20–3:

Table 20–3 Sample Disk I/O Analysis

Disk Speed	File System	Raw Devices
Reads per second	50	45
Writes per second	20	50

5. Calculate the number of disks you need to achieve your I/O performance target using a chart like the one shown in Table 20–4:

Table 20–4 Disk I/O Requirements Worksheet

	If Stored on File System			If Stored on Raw Devices		
Object	R/W Needed per Sec.	Disk R/W Capabil. per Sec.	Disks Needed	R/W Needed per Sec.	Disk R/W Capabil. per Sec.	Disks Needed
A						
B						
C						
Disks Req'd						

Table 20–5 shows the values from this example:

Table 20–5 Sample Disk I/O Requirements

	If Stored on File System			If Stored on Raw Devices		
Object	**R/W Needed per Sec.**	**Disk R/W Capabil. per Sec.**	**Disks Needed**	**R/W Needed per Sec.**	**Disk R/W Capabil. per Sec.**	**Disks Needed**
A	100 reads	50 reads	2 disks	100 reads	45 reads	3 disks
B	100 reads	50 reads	2 disks	100 reads	45 reads	3 disks
C	100 writes	20 writes	5 disks	100 writes	50 writes	2 disks
Disks Req'd			9 disks			8 disks

Planning File Storage

This section explains the following:

- How to determine the types of I/O operations required by your application.
- How to choose between file system and raw devices for your database files.

Design Approach

Use the following approach to design file storage:

1. Identify the operations required by your application.
2. Test the performance of your system's disks and controllers for the different operations required by your application.
3. Finally, evaluate what kind of disk and controller layout gives you the best performance for the operations that predominate in your application.

These steps are described in detail under the following headings.

Identifying the Required Read/Write Operations

Evaluate your application to determine how often it requires each type of I/O operation (sequential read, sequential write, random read, and random write).

Table 20–6 shows the types of read and write operations performed by each of the background processes, by foreground processes, and by parallel execution servers.

Table 20–6 Read/Write Operations Performed by Oracle Processes

	Process							
Operation	**LGWR**	**DBW*n***	**ARCH**	**SMON**	**PMON**	**CKPT**	**Foreground**	**PQ Processes**
Sequential Read			X	X		X	X	X
Sequential Write	X		X			X	X	X
Random Read				X			X	
Random Write		X						

In this discussion, a sample application might involve 50% random reads, 25% sequential reads, and 25% random writes.

Sequential I/O Sequential I/O is characterized by high data rates. For example, a single DSS type I/O may access hundreds of blocks. Sequential access is efficient, because these accesses allow data prefetches and cause limited head positioning. This provides high throughputs.

Because DSS systems may not do a large number of transactions per second, it is better to estimate the size of the I/O in terms of bytes per second. For example:

```
(estimate # of physical blocks in transaction * Oracle block size) = byte/second
```

Using this value and the theoretical limits for disk and controller throughputs can help you determine the number of drives/controllers to implement.

> **Note:** Most disk drives can handle 50-70 I/O per second and can transfer approximately 5Mb/sec.

The goal in optimizing sequential I/O is to maximize throughput by involving the maximum number of disks in the I/O request. The more disks involved, the greater aggregate throughput. For example:

```
4 disks/array @ 5Mb/second = (20 Mb/second)I/O call
```

Random I/O Random I/O is characterized by high I/O rate in OLTP. It requires frequent seeks with small I/O sizes.

The following example determines the application load for OLTP (gets the number and size of the transactions):

```
(# of blocks accessed/transaction) * (# of transactions/second) = blocks/second
```

Using this value and the theoretical limits for disk and controller throughputs can help you determine the number of drives per controllers to implement.

The goal in optimizing random I/O is to reduce disk hot spots and limit seek times.

Testing the Performance of Your Disks

This section illustrates relative performance of read/write operations by a particular test system.

> **Note:** Values provided in this example do *not* constitute a general rule. They were generated by an actual UNIX test system using particular disks. *These figures differ significantly for different platforms and different disks!* To make accurate judgments, *test your own system* using an approach similar to the one demonstrated in this section. Or, contact your system vendor for information on disk performance for the different operations.

Table 20–7 shows the speed of sequential read in milliseconds per I/O on a test system.

Table 20–7 Block Size and Speed of Sequential Read (Sample Data)

	Speed of Sequential Read on:	
Block Size	**Raw Device**	**UNIX File System (UFS)**
512 bytes	1.4	0.4
1KB	1.4	0.3
2KB	1.5	0.6
4KB	1.6	1.0
8KB	2.7	1.5
16KB	5.1	3.7
32KB	10.1	8.1
64KB	20.0	18.0
128KB	40.4	36.1
256KB	80.7	61.3

Doing research like this helps determine the correct stripe size. In this example, it takes at most 5.3 milliseconds to read 16KB. If your data is in chunks of 256KB, then you could stripe the data over 16 disks (as described on page 20-22) and maintain this low read time.

By contrast, if all your data is on one disk, then read time would be 80 milliseconds. Thus, the test results show that on this particular set of disks, things look quite different from what might be expected. It is sometimes beneficial to have a smaller stripe size, depending on the size of the I/O.

Table 20–8 shows the speed of sequential write in milliseconds per I/O on the test system.

Table 20–8 Block Size and Speed of Sequential Write (Sample Data)

	Speed of Sequential Write on:	
Block Size	**Raw Device**	**UNIX File System (UFS)**
512 bytes	11.2	17.9
1KB	11.7	18.3
2KB	11.6	19.0
4KB	12.3	19.8
8KB	13.5	21.8
16KB	16.0	35.3
32KB	19.3	62.2
64KB	31.5	115.1
128KB	62.5	221.8
256KB	115.6	429.0

Table 20–9 shows the speed of random read in milliseconds per I/O on the test system.

Table 20–9 Block Size and Speed of Random Read (Sample Data)

	Speed of Random Read on:	
Block Size	**Raw Device**	**UNIX File System (UFS)**
512 bytes	12.3	15.5
1KB	12.0	14.1
2KB	13.4	15.0
4KB	13.9	15.3
8KB	15.4	14.4
16KB	19.1	39.7
32KB	25.7	39.9
64KB	38.1	40.2
128KB	64.3	62.2
256KB	115.7	91.2

Table 20–10 shows the speed of random write in milliseconds per I/O on the test system.

Table 20–10 Block Size and Speed of Random Write (Sample Data)

	Speed of Random Write on:	
Block Size	**Raw Device**	**UNIX File System (UFS)**
512 bytes	12.3	40.7
1KB	12.0	41.4
2KB	12.6	41.6
4KB	13.8	41.4
8KB	14.8	32.8
16KB	17.7	45.6
32KB	24.8	71.6
64KB	38.0	123.8
128KB	74.4	230.3
256KB	137.4	441.5

Evaluate Disk Layout Options

Knowing the types of operations that predominate in your application and the speed with which your system can process the corresponding I/Os, you can choose the disk layout that maximizes performance.

For example, with the sample application and test system described previously, the UNIX file system is a good choice. With random reads predominating (50% of all I/O operations), 8KB is good block size. Furthermore, the UNIX file system in this example processes sequential reads (25% of all I/O operations) almost twice as fast as raw devices, given an 8KB block size.

Note: *Figures shown in the preceding example differ significantly on different platforms, and with different disks!* To plan effectively, test I/O performance on your own system.

Choosing Data Block Size

Table data in the database is stored in data blocks. This section describes how to allocate space within data blocks for best performance.

With single block I/O (random read), minimize the number of reads required to retrieve the desired data. How you store the data determines whether this performance objective is achieved. It depends on two factors: storage of the rows and block size.

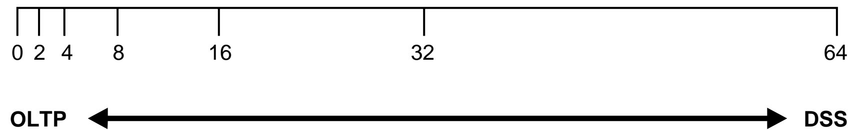
Tests have proven that matching the database block size to the UNIX file system (UFS) block size provides the most predictable and efficient performance. On UNIX systems, the block size of an existing file system can be determined using the `df -g` command.

Having the database block size greater than the UFS block size, or having the UFS block size greater than database block size, may yield inconsistent performance based on how your operating system and external I/O subsystem manage data pre-fetching and the coalescing of multiple I/Os.

See Also: For more information, see the example in "Testing the Performance of Your Disks" on page 20-7.

Figure 20–1 illustrates the suitability of various block sizes to online transaction processing (OLTP) or decision support (DSS) applications.

Figure 20–1 Block Size and Application Type



See Also: Your Oracle operating system-specific documentation has information on the minimum and maximum block size on your operating system

Block Size Advantages and Disadvantages

Table 20–11 lists the advantages and disadvantages of different block sizes.

Table 20–11 Block Size Advantages and Disadvantages

Block Size	Advantages	Disadvantages
Small (2KB-4KB)	Reduces block contention. Good for small rows, or lots of random access.	Has relatively large overhead. You may end up storing only a small number of rows, depending on the size of the row.
Medium (8KB)	If rows are medium size, then you can bring a number of rows into the buffer cache with a single I/O. With 2KB or 4KB block size, you may only bring in a single row.	Space in the buffer cache is wasted if you are doing random access to small rows and have a large block size. For example, with an 8KB block size and 50 byte row size, you are wasting 7,950 bytes in the buffer cache when doing random access.
Large (16KB-32KB)	There is relatively less overhead; thus, there is more room to store useful data. Good for sequential access or very large rows.	Large block size is not good for index blocks used in an OLTP type environment, because they increase block contention on the index leaf blocks.

Evaluating Device Bandwidth

The number of I/Os a disk can perform depends on whether the operations involve reading or writing to objects stored on raw devices or on the file system. This affects the number of disks you must use to achieve the desired level of performance.

I/O Tuning Tips

When performing I/O tuning, remember that I/O service time reported by the operating system is not necessarily the total time taken to process the I/O. The goal of tuning I/O is to minimize waits, such that the response time equals service time plus wait time.

An I/O queue consists of two stages: one stage is in the device driver, and the second stage is in the device itself. When the I/O is waiting to be serviced by the SCSI bus or disk, it is actually waiting in the device driver queue. This is the true wait queue. The time on the wait queue is the wait queue time. When the I/O is truly being service by the disk unit, then it is on the active or run queue. The time to physically process the I/O is the service time.

The goal in tuning I/O is to minimize waits and increase throughput. Disk time can encounter the following bottlenecks:

- Waiting for access to the I/O bus.
- Waiting for other queued disk requests.
- Seek time (to find the correct track in cylinder).
- Rotation time (to find the correct sector in track).
- I/O transfer time.

Tips for Tuning the I/O Adapter

- Spread disks over several I/O adapters, so that one I/O adapter is not overworked. Also, spread I/O bound files across several disks and several I/O adapters using appropriate RAID implementation. Striping allows single I/O requests to be serviced by several drives, and it allows multiple I/Os to occur in parallel.
- Make sure that there are not too many adapters on the same system bus. Exceeding the bandwidth of the system bus creates large CPU waits.
- Understand your applications. Make sure that the number of I/O operations (or the I/O size) of your application does not exceed the theoretical limit of the adapter or a given disk.

Dissecting the I/O Path

This section explains the I/O path, so that you can analyze and tune I/O bottlenecks. The I/O path follows these steps:

1. A user process issues an I/O call (read or write).

2. The I/O is placed on an available CPU’s dispatch queue. An available CPU picks up the request and context switches the user process.
3. The CPU checks the local cache to see if the requested data block is there. If it is, then it is a cache hit, and the I/O request is complete.
4. If the block does not exist in the cache or main memory, then it takes a major page fault (gets page from disk), and issues an I/O call against the appropriate device driver. The device driver builds a set of SCSI commands against an I/O unit.
5. The operating system (device driver) sends an I/O request through system bus to I/O controller (host bus adapter).
6. The host bus adapter (HBA) arbitrates for bus access. When the I/O request’s device is ready, it is selected, and the I/O statement is prepared to be sent to the target.
7. If the target unit can satisfy the request from its cache, then it transfers the data back and disconnects. This is a disk cache hit. For a cache miss, the target disconnects and tries to service the request.
8. The I/O request is placed in the target's queue table on its adapter where it may be sorted and merged with other I/O requests. This is possible only if the disk unit supports tag queuing.
9. After the I/O is picked off the queue, it is serviced by computing the physical address and seeking to the correct sector, read or write. If it is a read operation, then data is placed in the target cache. Write operations signal a completion by sending an interrupt signal.
10. The target controller reconnects with the I/O bus to transfer the data (with reads).
11. The HBA sends an interrupt to the operating system, marking the end of the I/O request.

The following table explains where the wait components lay with respect to the I/O path.

Steps 1 - 5, 11	These steps are handled by the operating system. The time required for these operations is limited by access time to the HBA. Slowness can be attributed to CPU contention or I/O bus contention. A heavily loaded CPU is not able to service an I/O request. Review `vmstat` statistics for runnable process, high system time, and excessive context switch.

Steps 5, 6, 10, 11 These two steps involve the I/O adapter. A faster adapter propagates and manages I/O requests faster. An overloaded I/O bus may cause I/O requests to process slowly. Review `IOSTAT` statistics for high percent busy combined with large `AVWAIT` (or `AVSERV`, if available).

Steps 7 - 9 These two steps are handled by the disk drive. Limiting factors can be seek times, rotational delays, and data transfer times. These disk operations are mechanical in nature; therefore, they consume the largest chunk of time in an I/O call. Newer disks have improved seek times, rotational speeds, and data transfer rates.

Mechanical time is considered wasted time, because no data (productive work) is transferred during this time. The goal is to minimize this time by acquiring disks with larger caches and by using disks with tag queuing.

To minimize the mechanical overhead of an I/O, spread the I/O request across several disks using an appropriate stripe size under a RAID implementation. An incorrect stripe size can cause hot disks or multiple physical I/Os per logical I/O.

Additionally, use a raw interface (raw devices) or direct I/O when possible. Raw devices allow unbuffered I/O and can utilize kernalized asynchronous I/O. Raw interfaces can also be implemented using a volume manager. Finally, check to see if your operating system provides direct I/O support on file system-based files. Direct I/O has proven to be helpful for I/Os that involve sequential reads and writes.

See Also: For more information on removing I/O contention, see "Reducing Disk Contention by Distributing I/O" on page 20-18.

Detecting I/O Problems

This section describes two tasks to perform if you suspect a problem with I/O usage:

- Checking System I/O Utilization
- Checking Oracle I/O Utilization

Oracle compiles file I/O statistics that reflect disk access to database files. These statistics report only the I/O utilization of Oracle sessions—yet every process affects the available I/O resources. Tuning non-Oracle factors can thus improve performance.

Checking System I/O Utilization

Use operating system monitoring tools to determine what processes are running on the system as a whole, and to monitor disk access to all files. Remember that disks holding datafiles and redo log files may also hold files that are not related to Oracle. Try to reduce any heavy access to disks that contain database files. Access to non-Oracle files can be monitored only through operating system facilities rather than through the `V$FILESTAT` view.

Tools, such as `sar -d`, on many UNIX systems let you examine the I/O statistics for your entire system. (Some UNIX-based platforms have an `iostat` command.) On NT systems, use Performance Monitor.

> **See Also:** For information on other platforms, see your operating system documentation.

Checking Oracle I/O Utilization

This section identifies the views and processes that provide Oracle I/O statistics. It also shows how to check statistics using `V$FILESTAT`.

Dynamic Performance Views for I/O Statistics

Table 20–12 shows dynamic performance views to check for I/O statistics relating to Oracle database files, log files, archive files, and control files.

Table 20–12 Where to Find Statistics about Oracle Files

File Type	Where to Find Statistics
Database Files	V$FILESTAT, V$SYSTEM_EVENT, V$SESSION_EVENT
Log Files	V$SYSSTAT, V$SYSTEM_EVENT, V$SESSION_EVENT
Archive Files	V$SYSTEM_EVENT, V$SESSION_EVENT
Control Files	V$SYSTEM_EVENT, V$SESSION_EVENT

Table 20–13 lists which file types processes write to.

Table 20–13 File Throughput Statistics for Oracle Processes

	Process							
File	LGWR	DBW*n*	ARCH	SMON	PMON	CKPT	Foreground	PQ Process
Database Files		X		X	X	X	X	X
Log Files	X							
Archive Files			X					
Control Files	X	X	X	X	X	X	X	X

V$SYSTEM_EVENT can be queried by event to show the total number of I/Os and average duration by type of I/O (read/write). With this, you can determine which types of I/O are too slow. If there are Oracle-related I/O problems, then tune them. But, if your process is not consuming the available I/O resources, then some other process is. Go back to the system to identify the process that is using up so much I/O, and determine why. Then tune this process.

Note: Different types of I/O in Oracle require different tuning approaches. Tuning I/O for data warehousing applications that perform large sequential reads is different from tuning I/O for OLTP applications that perform random reads and writes.

See Also: "Planning File Storage" on page 20-5.

Checking Oracle Datafile I/O with V$FILESTAT

Examine disk access to database files through the dynamic performance view V$FILESTAT. This view shows the following information for database I/O (but not for log file I/O):

- Number of physical reads and writes.
- Number of blocks read and written.
- Total I/O time for reads and writes.

By default, this view is available only to the user SYS and to users granted SELECT ANY TABLE system privilege, such as SYSTEM. The following column values reflect the number of disk accesses for each datafile:

PHYRDS	The number of reads from each database file.
PHYWRTS	The number of writes to each database file.

Use the following query to monitor these values over some period of time while your application is running:

```
SELECT NAME, PHYRDS, PHYWRTS
FROM V$DATAFILE df, V$FILESTAT fs
WHERE df.FILE# = fs.FILE#;
```

This query also retrieves the name of each datafile from the dynamic performance view V$DATAFILE. Sample output might look like this:

```
NAME                                            PHYRDS    PHYWRTS
-------------------------------------------- ---------- ----------
/oracle/ora70/dbs/ora_system.dbf                   7679       2735
/oracle/ora70/dbs/ora_temp.dbf                       32        546
```

The PHYRDS and PHYWRTS columns of V$FILESTAT can also be obtained through SNMP.

The total I/O for a single disk is the sum of PHYRDS and PHYWRTS for all the database files managed by the Oracle instance on that disk. Determine this value for each of your disks. Also, determine the rate at which I/O occurs for each disk by dividing the total I/O by the interval of time over which the statistics were collected.

Note: Although Oracle records read and write times accurately, a database that is running on UFS may not reflect true disk accesses. For example, the read times may not always reflect a true disk read, but rather a UFS cache hit. However, read and write times should be accurate for raw devices. Additionally, write times are only recorded per batch, with all blocks in the same batch given the same time after the completion of the write I/O.

Solving I/O Problems

The rest of this chapter describes various techniques of solving I/O problems:

- Reducing Disk Contention by Distributing I/O
- Striping Disks
- Avoiding Dynamic Space Management
- Tuning Sorts
- Tuning Checkpoint Activity
- Tuning LGWR and DBWR I/O
- Tuning Backup and Restore Operations
- Configuring the Large Pool

Reducing Disk Contention by Distributing I/O

This section describes how to reduce disk contention.

- What Is Disk Contention?
- Separating Datafiles and Redo Log Files
- Striping Table Data
- Separating Tables and Indexes
- Reducing Disk I/O Unrelated to Oracle

What Is Disk Contention?

Disk contention occurs when multiple processes try to access the same disk simultaneously. Most disks have limits on both the number of accesses and the

amount of data they can transfer per second. When these limits are reached, processes may have to wait to access the disk.

In general, consider the statistics in the `V$FILESTAT` view and your operating system facilities. Consult your hardware documentation to determine the limits on the capacity of your disks. Any disks operating at or near full capacity are potential sites for disk contention. For example, 60 or more I/Os per second may be excessive for some disks on VMS or UNIX operating systems.

In addition, review `V$SESSION_EVENT` for the following events: db file sequential read, db file scattered read, db file single write, and db file parallel write. These are all events corresponding to I/Os performed against the data file headers, control files, or data files. If any of these wait events correspond to high Average Time, then investigate the I/O contention using `sar` or `iostat`. Look for busy waits on the device. Examine the file statistics to determine which file is associated with the high I/O.

To reduce the activity on an overloaded disk, move one or more of its heavily accessed files to a less active disk. Apply this principle to each of your disks until they all have roughly the same amount of I/O. This is known as *distributing I/O*.

Separating Datafiles and Redo Log Files

Oracle processes constantly access datafiles and redo log files. If these files are on common disks, then there is potential for disk contention. Place each datafile on a separate disk. Multiple processes can then access different files concurrently without disk contention.

Place each set of redo log files on a separate disk with no other activity. Redo log files are written by the Log Writer process (LGWR) when a transaction is committed. Information in a redo log file is written sequentially. This sequential writing can take place much faster if there is no concurrent activity on the same disk. Dedicating a separate disk to redo log files usually ensures that LGWR runs smoothly with no further tuning attention. Performance bottlenecks related to LGWR are rare.

> **See Also:** For information on tuning LGWR, see the section "Detecting Contention for Redo Log Buffer Latches" on page 21-16.

Dedicating separate disks to datafiles and mirroring redo log files are important safety precautions. These steps ensure that the datafiles and the redo log files cannot both be lost in a single disk failure. Mirroring redo log files ensures that a redo log file cannot be lost in a single disk failure.

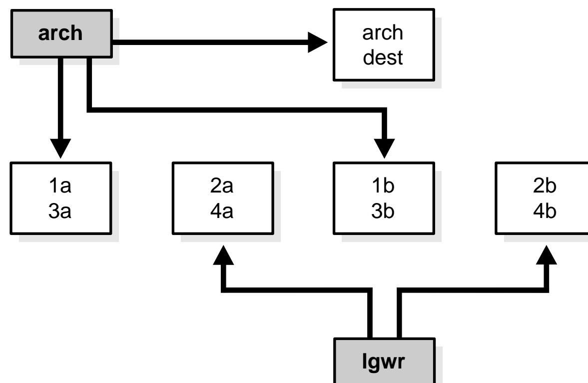
In order to prevent I/O contention between the archiver process and LGWR (when using multi-membered groups), make sure that archiver reads and LGWR writes are separated. For example, if your system has 4 groups with 2 members, then the following scenario should used to separate disk access:

4 groups x 2 members each = 8 logfiles labeled: 1a, 1b, 2a, 2b, 3a, 3b, 4a, 4b.

This requires at least 4 disks, plus one disk for archived files.

Figure 20–2 illustrates how redo members should be distributed across disks to minimize contention.

Figure 20–2 Distributing Redo Members Across Disks



In this example, LGWR switched out of log group 1 (member 1a and 1b) and is now writing to log group2 (2a and 2b). Concurrently, the archiver process is reading from the group 1 and writing to its archive destination. Note how the redo log files are isolated from contention.

Note: Mirroring redo log files, or maintaining multiple copies of each redo log file on separate disks, does not slow LGWR considerably. LGWR writes to each disk in parallel and waits until each part of the parallel write is complete. Because the time required to perform a single-disk write may vary, increasing the number of copies increases the likelihood that one of the single-disk writes in the parallel write takes longer than average. A parallel write does not take longer than the longest possible single-disk write. There may also be some overhead associated with parallel writes on your operating system.

Striping Table Data

Striping, or spreading a large table's data across separate datafiles on separate disks, can also help to reduce contention.

See Also: This strategy is fully discussed in the section "Striping Disks" on page 20-22.

Separating Tables and Indexes

It is not necessary to separate a frequently used table from its index. During the course of a transaction, the index is read first, and then the table is read. Because these I/Os occur sequentially, the table and index can be stored on the same disk without contention. However, for very high OLTP systems, separating indexes from tables may be required.

Split indexes and tables into separate tablespaces to minimize disk head movement and parallelize I/O. Both reads happen faster because one disk head is on the index data and the other is on the table data.

The idea of separating objects accessed simultaneously applies to indexes as well. For example, if a SQL statement uses two indexes at the same time, then performance is improved by having each index on a separate disk.

Also, avoid having several heavily accessed tables on the same disk. This requires strong knowledge of the application access patterns.

The use of partitioned tables and indexes can improve performance of operations in a data warehouse. Divide a large table or index into multiple physical segments residing in different tablespaces. All tables that contain large object datatypes should be placed into a separate tablespace as well.

Reducing Disk I/O Unrelated to Oracle

If possible, eliminate I/O unrelated to Oracle on disks that contain database files. This measure is especially helpful in optimizing access to redo log files. Not only does this reduce disk contention, it also allows you to monitor all activity on such disks through the dynamic performance table `V$FILESTAT`.

Striping Disks

This section describes:

- Purpose of Striping
- I/O Balancing and Striping
- Striping Disks Manually
- Striping Disks with Operating System Software
- Striping and RAID

Purpose of Striping

Striping divides a large table's data into small portions and stores these portions in separate datafiles on separate disks. This permits multiple processes to access different portions of the table concurrently without disk contention. Striping is particularly helpful in optimizing random access to tables with many rows. Striping can either be done manually (described below), or through operating system striping utilities.

I/O Balancing and Striping

Benchmark tuners in the past tried hard to ensure that the I/O load was evenly balanced across the available devices. Currently, operating systems are providing the ability to stripe a heavily used container file across many physical devices. However, such techniques are productive only where the load redistribution eliminates or reduces some form of queue.

If the wait service time exists, along with high percentage busy on a drive, then I/O distribution may be required. Where larger numbers of physical drives are available, consider dedicating two drives to carrying redo logs (two because redo logs should always be mirrored either by the operating system or using Oracle redo log group features). Because redo logs are written serially, drives dedicated to redo log activity normally require limited head movement. This significantly accelerates log writing.

When archiving, it is beneficial to use extra disks so that LGWR and ARCH do not compete for the same read/write head. This is achieved by placing logs on alternating drives.

Mirroring can also be a cause of I/O bottlenecks. The process of writing to each mirror is normally done in parallel, and does not cause a bottleneck. However, if each mirror is striped differently, then the I/O is not completed until the slowest mirror member is finished. To avoid I/O problems, stripe using the same number of disks for the destination database, or the copy, as you used for the source database.

For example, if you have 160KB of data striped over 8 disks, but the data is mirrored onto only one disk, then regardless of how quickly the data is processed on the 8 disks, the I/O is not completed until 160KB has been written onto the mirror disk. It might thus take 20.48 milliseconds to write the database, but 137 milliseconds to write the mirror.

Striping Disks Manually

To stripe disks manually, you need to relate an object's storage requirements to its I/O requirements.

1. Begin by evaluating an object's disk storage requirements by checking:

   - The size of the object.
   - The size of the disk.

   For example, if an object requires 5GB in Oracle storage space, then you need one 5GB disk or two 4GB disks to accommodate it. On the other hand, if the system is configured with 1GB or 2GB disks, then the object may require 5 or 3 disks, respectively.

2. Compare to this the application's I/O requirements, as described in "Analyzing I/O Requirements" on page 20-3. You must take the larger of the storage requirement and the I/O requirement.

   For example, if the storage requirement is 5 disks (1GB each), and the I/O requirement is 2 disks, then your application requires the higher value: 5 disks.

3. Create a tablespace with the `CREATE TABLESPACE` statement. Specify the datafiles in the `DATAFILE` clause. Each of the files should be on a different disk. For example:

```
CREATE TABLESPACE stripedtabspace
   DATAFILE 'file_on_disk_1' SIZE 1GB,
      'file_on_disk_2' SIZE 1GB,
```

```
'file_on_disk_3' SIZE 1GB,
'file_on_disk_4' SIZE 1GB,
'file_on_disk_5' SIZE 1GB;
```

4. Then, create the table with the `CREATE TABLE` statement. Specify the newly created tablespace in the `TABLESPACE` clause.

   Also specify the size of the table extents in the `STORAGE` clause. Store each extent in a separate datafile. The table extents should be slightly smaller than the datafiles in the tablespace to allow for overhead. For example, when preparing for datafiles of 1GB (1024MB), you can set the table extents to be 1023MB. For example:

```
CREATE TABLE stripedtab (
   col_1  NUMBER(2),
   col_2  VARCHAR2(10) )
   TABLESPACE stripedtabspace
   STORAGE ( INITIAL 1023MB  NEXT 1023MB
      MINEXTENTS 5  PCTINCREASE 0 );
```

   (Alternatively, you can stripe a table by entering an `ALTER TABLE ALLOCATE EXTENT` statement with `DATAFILE` '*datafile*' `SIZE` '*size*'.)

These steps result in the creation of table `STRIPEDTAB`. `STRIPEDTAB` has 5 initial extents, each of size 1023MB. Each extent takes up one of the datafiles named in the `DATAFILE` clause of the `CREATE TABLESPACE` statement. Each of these files is on a separate disk. The 5 extents are all allocated immediately, because `MINEXTENTS` is 5.

> **See Also:** For more information on `MINEXTENTS` and the other storage parameters, see *Oracle8i SQL Reference.*

Striping Disks with Operating System Software

As an alternative to striping disks manually, use operating system utilities or third-party tools, such as logical volume managers, or use hardware-based striping.

With utilities or hardware-based striping mechanisms, the main factors to consider are stripe size, number of disks to stripe across (which defines the stripe width), and the level of concurrency (or level of I/O activity). These factors are affected by the Oracle block size and the database access methods.

Table 20–14 Minimum Stripe Size

Disk Access	Minimum Stripe Size
Random reads and writes	The minimum stripe size is twice the Oracle block size.
Sequential reads	The minimum stripe size is twice the value of DB_FILE_MULTIBLOCK_READ_COUNT.

Table 20–15 Typical Stripe Size

Concurrency	I/O Size	Typical Stripe Size
Low	Small	k * DB_BLOCK_SIZE
Low	Large	k * DB_BLOCK_SIZE
High	Small	k * DB_BLOCK_SIZE
High	Large	k * DB_BLOCK_SIZE * DB_FILE_MULTI_BLOCK_READ_COUNT

Where k = 2,3,4...

In striping, uniform access to the data is assumed. If the stripe size is too large, then a hot spot may appear on one disk or on a small number of disks. Avoid this by reducing the stripe size, thus spreading the data over more disks.

Consider an example in which 100 rows of fixed size are evenly distributed over 5 disks, with each disk containing 20 sequential rows. If your application only requires access to rows 35 through 55, then only 2 disks must perform the I/O. At a high rate of concurrency, the system may not be able to achieve the desired level of performance.

Correct this problem by spreading rows 35 through 55 across more disks. In the current example, if there were two rows per block, then we could place rows 35 and 36 on the same disk, and rows 37 and 38 on a different disk. Taking this approach, we could spread the data over all the disks and I/O throughput would improve.

Striping and RAID

Redundant arrays of inexpensive disks (RAID) configurations provide improved data reliability. However, I/O performance depends on which RAID configuration is implemented.

Below are the most widely used RAID configurations:

- RAID 1: Provides good reliability and good read rates; however, writes may be costly.
- RAID 0+1: Provides good reliability and better read and write performance than RAID 1.
- RAID 5: Provides good reliability. Sequential reads benefit the most. Writes performance may suffer with RAID 5. This configuration is not recommended for heavy write applications.

Note: Although RAID 0 provides the best read and write performance, it is not a true RAID system, because it does not allow for redundancy. Oracle recommends that you do not place production database files on RAID 0 systems.

Optimal stripe size is a function of three things:

1. Size of I/O requests to the array.
2. Concurrency of I/O requests to the array.
3. The physical stripe boundaries matching the block size boundaries.

Striping is a good tool for balancing I/O across two or more disks in an array. However, keep in mind the following techniques:

- On high concurrency arrays, you must ensure that no single I/O request gets broken up into more than one physical I/O call. Failing to do this multiplies the number of physical I/O requests performed in your system, which in turn causes exponential degradation in your system I/O response times.
- On low concurrency arrays, you must ensure that no single I/O visits any disk twice. To fail here causes the same performance penalty as the one described above.

Avoiding Dynamic Space Management

When you create an object, such as a table or rollback segment, Oracle allocates space in the database for the data. This space is called a *segment*. If subsequent database operations cause the data volume to increase and exceed the space allocated, then Oracle extends the segment. Dynamic extension then reduces performance.

This section discusses:

- Detecting Dynamic Extension
- Allocating Extents
- Evaluating Unlimited Extents
- Evaluating Multiple Extents
- Avoiding Dynamic Space Management in Rollback Segments
- Reducing Migrated and Chained Rows
- Modifying the SQL.BSQ File
- Using Locally-Managed Tablespaces

Detecting Dynamic Extension

Dynamic extension causes Oracle to execute SQL statements in addition to those SQL statements issued by user processes. These SQL statements are called *recursive calls* because Oracle issues these statements itself. Recursive calls are also generated by these activities:

- Misses on the data dictionary cache.
- Firing of database triggers.
- Execution of Data Definition Language (DDL) statements.
- Execution of SQL statements within stored procedures, functions, packages, and anonymous PL/SQL blocks.
- Enforcement of referential integrity constraints.

Examine the `RECURSIVE CALLS` statistic through the dynamic performance view `V$SYSSTAT`. By default, this view is available only to user `SYS` and to users granted the `SELECT ANY TABLE` system privilege, such as `SYSTEM`. Use the following query to monitor this statistic over a period of time:

```
SELECT NAME, VALUE
FROM V$SYSSTAT
WHERE NAME = 'recursive calls';
```

Oracle responds with something similar to the following:

```
NAME                                                      VALUE
------------------------------------------------------- ----------
recursive calls                                           626681
```

If Oracle continues to make excessive recursive calls while your application is running, then determine whether these recursive calls are due to an activity, other than dynamic extension, that generates recursive calls. If you determine that the recursive calls are caused by dynamic extension, then reduce this extension by allocating larger extents.

Allocating Extents

Follow these steps to avoid dynamic extension:

1. Determine the maximum size of your object.
2. Choose storage parameter values so that Oracle allocates extents large enough to accommodate all your data when you create the object.

Larger extents tend to benefit performance for the following reasons:

- Blocks in a single extent are contiguous, so one large extent is more contiguous than multiple small extents. Oracle can read one large extent from disk with fewer multiblock reads than would be required to read many small extents. Therefore, make sure that the extent size is a multiple of `DB_FILE_MULTI_BLOCK_READ_COUNT`.
- Segments with larger extents are less likely to be extended.

However, because large extents require more contiguous blocks, Oracle may have difficulty finding enough contiguous space to store them. To determine whether to allocate only a few large extents or many small extents, evaluate the benefits and drawbacks of each in consideration of plans for the growth and use of your objects.

Automatically re-sizable datafiles can also cause problems with dynamic extension. Instead, manually allocate more space to a datafile during times when the system is relatively inactive.

Evaluating Unlimited Extents

Even though an object may have unlimited extents, this does not mean that having a large number of small extents is acceptable. For optimal performance you may decide to reduce the number of extents.

Extent maps list all extents for a particular segment. The number of extents entries per Oracle block depends on operating system block size and platform. Although an extent is a data structure inside Oracle, the size of this data structure depends on the platform. Accordingly, this affects the number of extents Oracle can store in a single operating system block. Typically, this value is as follows:

Table 20–16 Block Size and Maximum Number of Extents (Typical Values)

Block Size (KB)	Maximum Number of Extents
2	121
4	255
8	504
16	1032
32	2070

For optimal performance, you should be able to read the extent map with a single I/O. Performance degrades if multiple I/Os are necessary for a full table scan to get the extent map.

Avoid dynamic extension in dictionary-mapped tablespaces. For dictionary-mapped tablespaces, do not let the number of extents exceed 1,000. If extent allocation is local, then do not have more than 2,000 extents. Having too many extents reduces performance when dropping or truncating tables.

Evaluating Multiple Extents

This section explains various ramifications of using multiple extents.

- You cannot put large segments into single extents, because of file size and file system size limitations. When you enable segments to allocate new extents over time, you can take advantage of faster, less expensive disks.
- For a table that is never full-table scanned, it makes no difference in terms of query performance whether the table has one extent or multiple extents.
- The performance of searches using an index is not affected by the index having one extent or multiple extents.
- Using more than one extent in a table, cluster, or temporary segment does not affect the performance of full scans on a multi-user system.
- Using more than one extent in a table, cluster, or temporary segment does not materially affect the performance of full scans on a dedicated single-user batch processing system, if the extents are properly sized and if the application is designed to avoid expensive DDL operations.
- If extent sizes are appropriately matched to the I/O size, then the performance cost of having many extents in a segment is minimized.

- For rollback segments, many extents are preferable to few extents. Having many extents reduces the number of recursive SQL calls to perform dynamic extent allocations on the segments.

Avoiding Dynamic Space Management in Rollback Segments

The size of rollback segments can affect performance. Rollback segment size is determined by the rollback segment's storage parameter values. Your rollback segments must be large enough to hold the rollback entries for your transactions. As with other objects, you should avoid dynamic space management in rollback segments.

Use the `SET TRANSACTION` statement to assign transactions to rollback segments of the appropriate size based on the recommendations in the following sections. If you do not explicitly assign a transaction to a rollback segment, then Oracle automatically assigns it to a rollback segment.

For example, the following statement assigns the current transaction to the rollback segment `OLTP_13`:

```
SET TRANSACTION USE ROLLBACK SEGMENT oltp_13
```

Note: If you are running multiple concurrent copies of the same application, then be careful not to assign the transactions for all copies to the same rollback segment. This leads to contention for that rollback segment.

Also, monitor the shrinking, or dynamic deallocation, of rollback segments based on the `OPTIMAL` storage parameter.

> **See Also:** For information on choosing values for this parameter, monitoring rollback segment shrinking, and adjusting the `OPTIMAL` parameter, see *Oracle8i Administrator's Guide.*

For Long Queries Assign large rollback segments to transactions that modify data that is concurrently selected by long queries. Such queries may require access to rollback segments to reconstruct a read-consistent version of the modified data. The rollback segments must be large enough to hold all the rollback entries for the data while the query is running.

For Long Transactions Assign large rollback segments to transactions that modify large amounts of data. A large rollback segment can improve the performance of

such a transaction, because the transaction generates large rollback entries. If a rollback entry does not fit into a rollback segment, then Oracle extends the segment. Dynamic extension reduces performance and should be avoided whenever possible.

For OLTP Transactions OLTP applications are characterized by frequent concurrent transactions, each of which modifies a small amount of data. Assign OLTP transactions to small rollback segments, provided that their data is not concurrently queried. Small rollback segments are more likely to remain stored in the buffer cache where they can be accessed quickly. A typical OLTP rollback segment might have 2 extents, each approximately 10 kilobytes in size. To best avoid contention, create many rollback segments and assign each transaction to its own rollback segment.

Reducing Migrated and Chained Rows

If an `UPDATE` statement increases the amount of data in a row so that the row no longer fits in its data block, then Oracle tries to find another block with enough free space to hold the entire row. If such a block is available, then Oracle moves the entire row to the new block. This is called *migrating* a row. If the row is too large to fit into any available block, then Oracle splits the row into multiple pieces and stores each piece in a separate block. This is called *chaining* a row. Rows can also be chained when they are inserted.

Dynamic space management, especially migration and chaining, is detrimental to performance:

- `UPDATE` statements that cause migration and chaining perform poorly.
- Queries that select migrated or chained rows must perform more I/O.

Identify migrated and chained rows in a table or cluster using the `ANALYZE` statement with the `LIST CHAINED ROWS` clause. This statement collects information about each migrated or chained row and places this information into a specified output table.

The definition of a sample output table named `CHAINED_ROWS` appears in a SQL script available on your distribution medium. The common name of this script is `UTLCHN1.SQL`, although its exact name and location varies depending on your platform. Your output table must have the same column names, datatypes, and sizes as the `CHAINED_ROWS` table.

You can also detect migrated or chained rows by checking the `TABLE FETCH CONTINUED ROW` column in `V$SYSSTAT`. Increase `PCTFREE` to avoid migrated rows. If you leave more free space available in the block, then the row has room to

grow. You can also reorganize or re-create tables and indexes with high deletion rates.

Note: PCTUSED is not the opposite of PCTFREE; PCTUSED controls space management.

See Also: For more information on PCTUSED, see *Oracle8i Concepts*

To reduce migrated and chained rows in an existing table, follow these steps:

1. Use the ANALYZE statement to collect information about migrated and chained rows. For example:

   ```
   ANALYZE TABLE order_hist LIST CHAINED ROWS;
   ```

2. Query the output table:

   ```
   SELECT *
   FROM CHAINED_ROWS
   WHERE TABLE_NAME = 'ORDER_HIST';

   OWNER_NAME  TABLE_NAME  CLUST... HEAD_ROWID          TIMESTAMP
   ----------  ----------  -----... ------------------  ---------
   SCOTT       ORDER_HIST       ... AAAAluAAHAAAAA1AAA  04-MAR-96
   SCOTT       ORDER_HIST       ... AAAAluAAHAAAAA1AAB  04-MAR-96
   SCOTT       ORDER_HIST       ... AAAAluAAHAAAAA1AAC  04-MAR-96
   ```

   The output lists all rows that are either migrated or chained.

3. If the output table shows that you have many migrated or chained rows, then you can eliminate migrated rows with the following steps:

   a. Create an intermediate table with the same columns as the existing table to hold the migrated and chained rows:

      ```
      CREATE TABLE int_order_hist
         AS SELECT *
            FROM order_hist
            WHERE ROWID IN
               (SELECT HEAD_ROWID
                  FROM CHAINED_ROWS
                  WHERE TABLE_NAME = 'ORDER_HIST');
      ```

b. Delete the migrated and chained rows from the existing table:

```
DELETE FROM order_hist
   WHERE ROWID IN
      (SELECT HEAD_ROWID
         FROM CHAINED_ROWS
         WHERE TABLE_NAME = 'ORDER_HIST');
```

c. Insert the rows of the intermediate table into the existing table:

```
INSERT INTO order_hist
    SELECT *
    FROM int_order_hist;
```

d. Drop the intermediate table:

```
DROP TABLE int_order_history;
```

4. Delete the information collected in step 1 from the output table:

```
DELETE FROM CHAINED_ROWS
    WHERE TABLE_NAME = 'ORDER_HIST';
```

5. Use the ANALYZE statement again and query the output table.
6. Any rows that appear in the output table are chained. You can eliminate chained rows only by increasing your data block size. It may not be possible to avoid chaining in all situations. Chaining is often unavoidable with tables that have a LONG column or long CHAR or VARCHAR2 columns.

Retrieval of migrated rows is resource intensive; therefore, all tables subject to UPDATE should have their distributed free space set to allow enough space within the block for the likely update.

Modifying the SQL.BSQ File

The SQL.BSQ file runs when you issue the CREATE DATABASE statement. This file contains the table definitions that make up the Oracle server. The views you use as a DBA are based on these tables. Oracle recommends that you strictly limit modifications to SQL.BSQ.

- If necessary, you can increase the value of the following storage parameters: INITIAL, NEXT, MINEXTENTS, MAXEXTENTS, PCTINCREASE, FREELISTS, FREELIST GROUPS, and OPTIMAL.

> **See Also:** For complete information about these parameters, see *Oracle8i SQL Reference.*

- With the exception of `PCTINCREASE`, do not decrease the setting of a storage parameter to a value below the default. (If the value of `MAXEXTENTS` is large, then you can lower the value for `PCTINCREASE` or even set it to zero.)
- No other changes to `SQL.BSQ` are supported. In particular, you should not add, drop, or rename a column.

> **Note:** Oracle may add, delete, or change internal data dictionary tables from release to release. For this reason, modifications you make are not carried forward when the dictionary is migrated to later releases.

Using Locally-Managed Tablespaces

A tablespace that manages its own extents maintains a bitmap in each datafile to keep track of the free or used status of blocks in that datafile. Each bit in the bitmap corresponds to a block or a group of blocks. When an extent is allocated or freed for reuse, Oracle changes the bitmap values to show the new status of the blocks. These changes do not generate rollback information, because they do not update tables in the data dictionary (except for special cases such as tablespace quota information).

Locally-managed tablespaces have the following advantages over dictionary-managed tablespaces:

- Local management of extents avoids recursive space management operations, which can occur in dictionary-managed tablespaces if consuming or releasing space in an extent results in another operation that consumes or releases space in a rollback segment or data dictionary table.
- Local management of extents automatically tracks adjacent free space, eliminating the need to coalesce free extents.

The sizes of extents that are managed locally can be determined automatically by the system. Alternatively, all extents can have the same size in a locally-managed tablespace.

> **See Also:** For more information on locally-managed tablespaces, see *Oracle8i Concepts* and *Oracle8i Administrator's Guide.* For more information on the statements for specifying space management, see *Oracle8i SQL Reference.*

Tuning Sorts

There is a trade-off between performance and memory usage. For best performance, most sorts should occur in memory; sorts written to disk adversely affect performance. If the sort area size is too large, then too much memory may be used. If the sort area size is too small, then sorts may need to be written to disk which, as, mentioned, can severely degrade performance.

This section describes:

- Sorting to Memory
- Sorting to Disk
- Optimizing Sort Performance with Temporary Tablespaces
- Using NOSORT to Create Indexes Without Sorting
- GROUP BY NOSORT

Sorting to Memory

The default sort area size is adequate to hold all the data for most sorts. However, if your application often performs large sorts on data that does not fit into the sort area, then you may want to increase the sort area size. Large sorts can be caused by any SQL statement that performs a sort on many rows.

> **See Also:** For a list of SQL statements that perform sorts, see *Oracle8i Concepts.*

Recognizing Large Sorts Oracle collects statistics that reflect sort activity and stores them in the dynamic performance view `V$SYSSTAT`. By default, this view is available only to the user `SYS` and to users granted the `SELECT ANY TABLE` system privilege. These statistics reflect sort behavior:

`SORTS(MEMORY)`	The number of sorts small enough to be performed entirely in sort areas without I/O to temporary segments on disk.
`SORTS(DISK)`	The number of sorts too large to be performed entirely in the sort area, requiring I/O to temporary segments on disk.

Use the following query to monitor these statistics over time:

```
SELECT NAME, VALUE
FROM V$SYSSTAT
WHERE NAME IN ('SORTS (MEMORY)', 'SORTS (DISK)');
```

The output of this query might look like this:

```
NAME                                                       VALUE
-------------------------------------------------------- ----------
SORTS(MEMORY)                                                965
SORTS(DISK)                                                    8
```

The information in V$SYSSTAT can also be obtained through the Simple Network Management Protocol (SNMP).

Increasing SORT_AREA_SIZE to Avoid Sorting to Disk SORT_AREA_SIZE is a dynamically modifiable initialization parameter that specifies the maximum amount of memory to use for each sort. If a significant number of sorts require disk I/O to temporary segments, then your application's performance may benefit from increasing the size of the sort area. In this case, increase the value of SORT_AREA_SIZE.

The maximum value of this parameter depends on your operating system. You need to determine what size SORT_AREA_SIZE makes sense. If you set SORT_AREA_SIZE to an adequately large value, then most sorts should not need to go to disk (unless, for example, you are sorting a 10-gigabyte table).

> **See Also:** For more information, see the "Tuning Sort Areas" section in Chapter 19, "Tuning Memory Allocation".

Performance Benefits of Large Sort Areas As mentioned, increasing sort area size decreases the chances that sorts go to disk. Therefore, with a larger sort area, most sorts process quickly without I/O.

When Oracle writes sort operations to disk, it writes out partially sorted data in sorted runs. After all the data has been received by the sort, Oracle merges the runs to produce the final sorted output. If the sort area is not large enough to merge all the runs at once, then subsets of the runs are merged in several merge passes. If the sort area is larger, then there are fewer, longer runs produced. A larger sort area also means the sort can merge more runs in one merge pass.

Performance Trade-offs for Large Sort Areas Increasing sort area size causes each Oracle sort process to allocate more memory. This increase reduces the amount of memory for private SQL and PL/SQL areas. It can also affect operating system memory allocation and may induce paging and swapping. Before increasing the size of the sort area, be sure enough free memory is available on your operating system to accommodate a larger sort area.

If you increase sort area size, then consider decreasing the value for the SORT_AREA_RETAINED_SIZE parameter. This parameter controls the lower limit to which Oracle reduces the size of the sort area when Oracle completes some or all of a sort process. That is, Oracle reduces the size of the sort area after the sort has started sending the sorted data to the user or to the next part of the query. A smaller retained sort area reduces memory usage but causes additional I/O to write and read data to and from temporary segments on disk.

Sorting to Disk

Sort writes to disk directly bypass the buffer cache. If you sort to disk, then make sure that PCTINCREASE is set to zero for the tablespace used for sorting. Also, INITIAL and NEXT should be the same size. This reduces fragmentation of the tablespaces used for sorting. You set these parameters using the STORAGE clause of ALTER TABLE.

> **See Also:** For more information on PCTINCREASE, see *Oracle8i Concepts.*

Optimizing Sort Performance with Temporary Tablespaces

Optimize sort performance by performing sorts in temporary tablespaces. To create temporary tablespaces, use the CREATE TABLESPACE or ALTER TABLESPACE statements with the TEMPORARY keyword.

Normally, a sort may require many space allocation calls to allocate and deallocate temporary segments. If you specify a tablespace as TEMPORARY, then Oracle caches one sort segment in that tablespace for each instance requesting a sort operation. This scheme bypasses the normal space allocation mechanism and greatly improves performance of medium-sized sorts that cannot be done completely in memory.

You cannot use the TEMPORARY keyword with tablespaces containing permanent objects such as tables or rollback segments.

> **See Also:** For more information about the syntax of the CREATE TABLESPACE and ALTER TABLESPACE statements, see *Oracle8i SQL Reference.*

Striping Temporary Tablespaces Stripe the temporary tablespace over many disks, preferably using an operating system striping tool. For example, if you only stripe the temporary tablespace over 2 disks with a maximum of 50 I/Os per second on each disk, then Oracle can only perform 100 I/Os per second. This restriction could lengthen the duration of sort operations.

For the previous example, you could accelerate sort processing fivefold if you striped the temporary tablespace over 10 disks. This would enable 500 I/Os per second.

Using SORT_MULTIBLOCK_READ_COUNT Another way to improve sort performance using temporary tablespaces is to tune the parameter SORT_MULTIBLOCK_READ_COUNT. For temporary segments, SORT_MULTIBLOCK_READ_COUNT has nearly the same effect as the parameter DB_FILE_MULTIBLOCK_READ_COUNT.

Increasing the value of SORT_MULTIBLOCK_READ_COUNT forces the sort process to read a larger section of each sort run from disk to memory during each merge pass. This also forces the sort process to reduce the merge width, or number of runs, that can be merged in one merge pass. This may increase in the number of merge passes.

Because each merge pass produces a new sort run to disk, an increase in the number of merge passes causes an increase in the total amount of I/O performed during the sort. Carefully balance increases in I/O throughput obtained by increasing the SORT_MULTIBLOCK_READ_COUNT parameter with possible increases in the total amount of I/O performed.

Using NOSORT to Create Indexes Without Sorting

One cause of sorting is the creation of indexes. Creating an index for a table involves sorting all rows in the table based on the values of the indexed columns. Oracle also allows you to create indexes without sorting. If the rows in the table are loaded in ascending order, then you can create the index faster without sorting.

The NOSORT Clause To create an index without sorting, load the rows into the table in ascending order of the indexed column values. Your operating system may provide a sorting utility to sort the rows before you load them. When you create the index, use the NOSORT clause on the CREATE INDEX statement. For example, this CREATE INDEX statement creates the index EMP_INDEX on the ENAME column of the emp table without sorting the rows in the EMP table:

```
CREATE INDEX emp_index
   ON emp(ename)
   NOSORT;
```

Note: Specifying NOSORT in a CREATE INDEX statement negates the use of PARALLEL INDEX CREATE, even if PARALLEL (DEGREE *n*) is specified.

When to Use the NOSORT Clause Presorting your data and loading it in order may not always be the fastest way to load a table.

- If you have a multiple-CPU computer, then you may be able to load data faster using multiple processors in parallel, each processor loading a different portion of the data. To take advantage of parallel processing, load the data without sorting it first. Then create the index *without* the `NOSORT` clause.
- If you have a single-CPU computer, then you should sort your data before loading, if possible. Then create the index *with* the `NOSORT` clause.

GROUP BY NOSORT

Sorting can be avoided when performing a `GROUP BY` operation when you know that the input data is already ordered, so that all rows in each group are clumped together. This may be the case if the rows are being retrieved from an index that matches the grouped columns, or if a sort-merge join produces the rows in the right order. `ORDER BY` sorts can be avoided in the same circumstances. When no sort takes place, the `EXPLAIN PLAN` output indicates `GROUP BY NOSORT`.

Tuning Checkpoint Activity

Checkpointing is an operation that Oracle performs automatically. This section explains the following:

- How Checkpoints Affect Performance
- Adjusting Checkpointing Activity
- Fast-Start Checkpointing

> **See Also:** For a complete discussion of checkpoints, see *Oracle8i Concepts*.

How Checkpoints Affect Performance

Aggressive checkpointing will write dirty buffers to the datafiles more quickly and can reduce instance recovery time in the event of an instance failure. If checkpointing is fairly aggressive, then replaying the redo records in the redo log between the current checkpoint position and the end of the log involves processing relatively few data blocks. This means that the roll-forward phase of recovery will be fairly short.

However, aggressive checkpointing can reduce run-time performance, because checkpointing causes DBW*n* processes to perform I/O. The overhead associated with checkpointing is usually small.

Adjusting Checkpointing Activity

Adjust your checkpointing activity based on your performance concerns. If you are more concerned with efficient run-time performance than recovery time, then set checkpointing to be less aggressive.

If you are more concerned with having fast instance recovery than with achieving optimal run-time performance, then increase the checkpointing interval.

Checkpointing behavior can be influenced by the following parameters:

- Set the value of the `LOG_CHECKPOINT_INTERVAL` initialization parameter (in multiples of physical block size) to be larger than the size of your largest redo log file.
- Set the value of the `LOG_CHECKPOINT_TIMEOUT` initialization parameter to zero. This value eliminates time-based checkpoints.
- Set the value of `FAST_START_IO_TARGET` to zero to disable fast-start checkpointing. This is described below under the heading, "Fast-Start Checkpointing".

In addition to setting these parameters, also consider the size of your log files. Maintaining small log files can increase checkpoint activity and reduce performance.

Fast-Start Checkpointing

The fast-start checkpointing feature limits the number of dirty buffers and thereby limits the amount of time required for instance recovery. If Oracle must process an excessive number of I/O operations to perform instance recovery, then performance can be adversely affected. You can control this overhead by setting an appropriate value for the parameter `FAST_START_IO_TARGET`.

Note: Fast-start checkpointing is only available with the Oracle8*i* Enterprise Edition.

Oracle recommends using fast-start checkpointing to control the duration of the "roll-forward" phase of recovery. This behavior is controlled by the `FAST_START_IO_TARGET` parameter. The parameter, `DB_BLOCK_MAX_DIRTY_TARGET`, is an Oracle8 parameter used to provide more limited control over roll-forward duration, and it is included in Oracle8i only for backward compatibility.

`FAST_START_IO_TARGET` limits the number of I/O operations that Oracle should allow for instance recovery. If the number of operations required for recovery at any point in time exceeds this limit, then Oracle writes dirty buffers to disk until the number of I/O operations needed for instance recovery is reduced to the limit set by `FAST_START_IO_TARGET`.

You can control the duration of instance recovery, because the number of operations required to recover indicates how much time this process takes. Disable this aspect of checkpointing by setting `FAST_START_IO_TARGET` to zero (0).

> **See Also:** For more information on the `FAST_START_IO_TARGET` parameter and the trade-off between performance and instance recovery time, see Chapter 24, "Tuning Instance Recovery Performance".

Tuning LGWR and DBWR I/O

This section describes how to tune I/O for the log writer and database writer background processes.

Tuning LGWR I/O

Applications with many `INSERTs` or with `LONG/RAW` activity may benefit from tuning LGWR I/O. The size of each I/O write depends on the size of the log buffer which is set by the initialization parameter `LOG_BUFFER`. Therefore, it is important to choose the right log buffer size. LGWR starts writing if the buffer is one third full, or when it is posted by a foreground process such as a `COMMIT`. Too large a log buffer size might delay the writes. Too small a log buffer might also be inefficient, resulting in frequent, small I/Os.

If the average size of the I/O becomes quite large, then the log file could become a bottleneck. To avoid this problem, you can stripe the redo log files, going in parallel to several disks. You must use an operating system striping tool, because manual striping is not possible in this situation.

Stripe size is also important. You can figure an appropriate value by dividing the average redo I/O size by the number of disks over which you want to stripe the buffer.

Review `V$SYSSTAT` or the `UTLBSTAT` report for the following:

- Log buffer space: This is time spent waiting for space in the log buffer. This is an indication that the buffers are being filled up faster than LGWR is writing. This may also indicate disk I/O contention. If the count is very high, then increase LGWR buffers and investigate disk I/O contention where the redo logs reside.
- Log file space/switch: This is the time Oracle spent waiting for the space on disk for LGWR to complete the write of log buffers to the redo log. This may be an indication to increase log buffers.

For example:

```
SELECT a.VALUE / DECODE(b.VALUE, 0, 1, b.VALUE)
FROM V$SYSSTAT a, V$SYSSTAT b
WHERE a.NAME = 'redo size' AND b.NAME = 'user commits';
```

This provides the average number redo records per commit. Now you must determine the average number of commits per second and multiply it by the average number redo records per commit (calculated above). This provides the max log buffers to setup.

The following events uniquely identify the specific wait:

- Log buffer space
- Log file switch (checkpoint incomplete)
- Log file switch (archiving needed)
- Log file switch (clearing log file)
- Log file switch completion
- Switch logfile statement

Pre-Oracle 7.3 tuning required the following to tune log file switch counts:

- If log file/switch = log space free requests, then add more log_buffers

- If log file/switch is high and there is a significant difference between background checkpoints started and background checkpoints completed, then consider reviewing the checkpoint frequency and the log file sizes.

Tuning DBWR I/O

This section describes the following issues of tuning DBWR I/O:

- Multiple Database Writer (DBWR) Processes and Database Slaves
- Internal Write Batch Size
- LRU Latches with a Single Buffer Pool
- LRU Latches with Multiple Buffer Pools

Multiple Database Writer (DBWR) Processes and Database Slaves Multiple database writer processes are useful when a buffer cache is so large that one DBW*n* process running full-time cannot keep up with the load. However, for large transaction rate systems that have many CPUs, you can enable multiple database writers to handle the load.

Using the `DB_WRITER_PROCESSES` initialization parameter, you can create multiple database writer processes (from DBW0 to DBW9). Database I/O slaves provide non-blocking, asynchronous requests to simulate asynchronous I/O.

I/O slaves for DBWR are allocated immediately following database open, when the first I/O request is made. The DBWR continues to do all the DBWR-related work (scanning LRU). When the DBWR process initiates the I/O, the DBWR I/O slave simply does the I/O on behalf of DBWR. The writing of the batch is parallelized between the I/O slaves.

The main DBWR process, which is I/O issuing process, looks for an idle I/O slave. If one is available, then that I/O slave gets a post. If there are no idle slaves, then the I/O issuer spawns one. If the allowed number of slaves have been spawned, then the issuer waits and tries again to find an idle slave.

If the asynchronous I/O code of the platform has bugs or is not efficient, then asynchronous I/O can be disabled on a device type. However, multiple I/O slaves only parallelize the writing of the batch between the DBWR I/O slaves. In contrast, you can parallelize the gathering as well as the writing of buffers with the multiple DBWR feature. Therefore, from the throughput standpoint, N DBWR processes should deliver more throughput than one DBWR process with the same number of I/O slaves.

Multiple writer processes (and I/O slaves) are advanced features that are intended for heavy OLTP processing. Implement this feature only if the database

environment requires such I/O throughput. For example, if asynchronous I/O is available, then it may be wise to disable I/O slaves and run with a single DBWR in asynchronous I/O mode.

> **Note:** Review the current throughput, and examine possible bottlenecks to determine if it is feasible to implement these features.

If it is determined that there is a need for multiple writer processes or slave processes, then determine which option to use. Although both implementations of DBWR processes may be beneficial, the general indicator rule on which option to use depends on the availability of asynchronous I/O (from the operating system) and the number of CPUs.

The number of CPUs is also indirectly related to the number LRU latch sets. To determine whether to use multiple DBW*n* processes or database slaves, follow these guidelines:

- First, determine if DBWR is keeping up the write requests. Review the `V$SYSTEM_EVENT` view for significant numbers of 'free buffer' waits. Large values may indicate that users wants to read a buffer, but they cannot because there are too many dirty buffers in the cache. If you do not see free buffer waits, then DBWR is not a problem.
- If DBWR is keeping up with the write requests, then use I/O slaves only if asynchronous I/O is not supported (or is working improperly) on the operating system for the types of files that you are using. Start with the number of I/O slaves equal to the average number of disks that a typical file spans.
- If you are using a RAID device with a large write back cache, then you can reduce your I/O slaves significantly, because I/Os to the RAID cache are much faster than I/Os to the disks.
- Use multiple DBWR (`DB_WRITER_PROCESSES`) when one DBWR cannot keep up. This is generally only necessary for large SMP systems with a lot of I/O activity. Typically, there should be no more than one DBWR for every 8 CPUs. Some systems can use multiple DBWR processes with asynchronous I/O enabled if they have very high transaction rates, are not CPU starved, and the operating system's async I/O works effectively.

Note: Implementing `DB_IO_SLAVES` or multiple writer processes creates some overhead cost. Enabling these features requires that extra shared memory be allocated for I/O buffers and request queues.

Internal Write Batch Size Database writer (DBW*n)* processes use the *internal write batch size*, which is set to the *lowest* of the following three values (A, B, or C):

- Value A is calculated as follows:

$$\frac{DB_FILES * DB_FILE_SIMULTANEOUS_WRITES}{2} = Value\ A$$

- Value B is the port-specific limit. (See your Oracle platform-specific documentation.)
- Value C is one-fourth the value of `DB_BLOCK_BUFFERS`.

Setting the internal write batch size too large may result in uneven response times.

For best results, you can influence the internal write batch size by changing the parameter values by which Value A is calculated. Take the following approach:

1. Determine the files to which you must write, and the number of disks on which those files reside.
2. Determine the number of I/Os you can perform against these disks.
3. Determine the number of writes that your transactions require.
4. Make sure that you have enough disks to sustain this rate.

LRU Latches with a Single Buffer Pool When you have multiple database writer DBW*n* processes and only one buffer pool, the buffer cache is divided up among the processes by LRU (least recently used) latches; each LRU latch is for one LRU list.

The default value of the `DB_BLOCK_LRU_LATCHES` parameter is 50% of the number of CPUs in the system. You can adjust this value to be equal to, or a multiple of, the number of CPUs. The objective is to cause each DBW*n* process to have the same number of LRU lists, so that they have equivalent loads.

For example, if you have 2 database writer processes and 4 LRU lists (4 latches), then the DBW*n* processes obtain latches in a round-robin fashion. DBW0 obtains latch 1, DBW1 obtains latch 2, then DBW0 obtains latch 3 and DBW1 obtains latch 4.

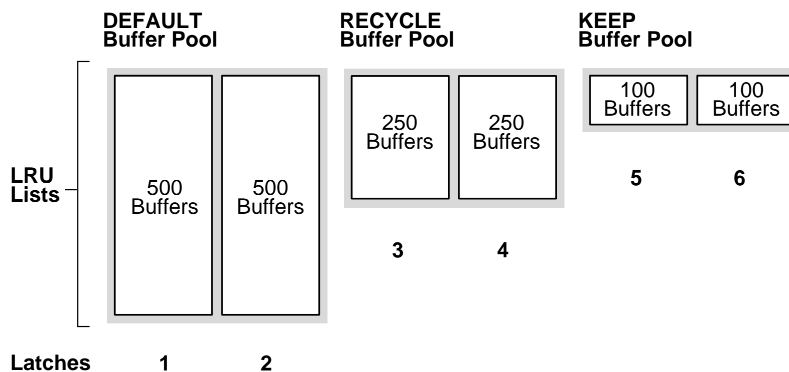
Similarly, if your system has 8 CPUs and 3 DBW*n* processes, then you should have 9 latches.

LRU Latches with Multiple Buffer Pools If you are using multiple buffer pools and multiple database writer (DBW*n*) processes, then the number of latches in each pool (`DEFAULT`, `KEEP`, and `RECYCLE`) should be equal to, or a multiple of, the number of processes. This is recommended so that each DBW*n* process is equally loaded.

> **Note:** When there are multiple buffer pools, each buffer pool has a contiguous range of LRU latches.

Consider the example in Figure 20–3 where there are 3 DBW*n* processes and 2 latches for each of the 3 buffer pools, for a total of 6 latches. Each buffer pool would obtain a latch in round robin fashion.

Figure 20–3 LRU Latches with Multiple Buffer Pools: Example 1



The `DEFAULT` buffer pool has 500 buffers for each LRU list. The `RECYCLE` buffer pool has 250 buffers for each LRU list. The `KEEP` buffer pool has 100 buffers for each LRU.

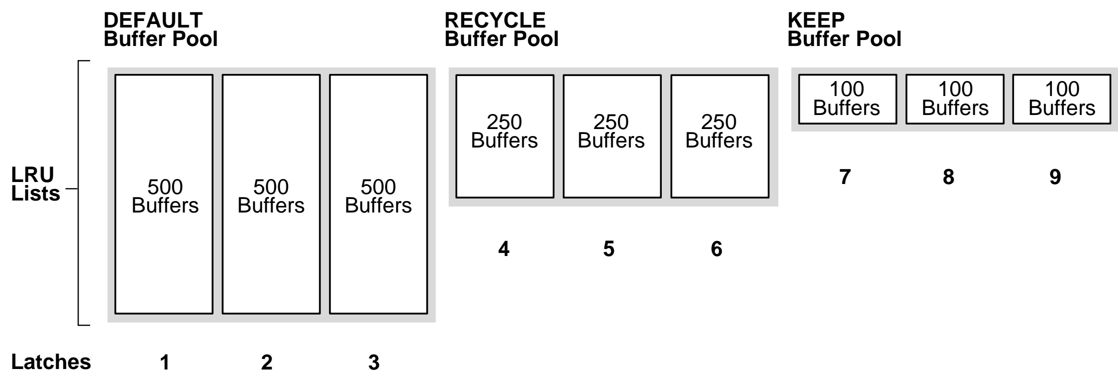
DBW0 gets latch 1 (500) and latch 4 (250) for 750.
DBW1 gets latch 2 (500) and latch 6 (100) for 600.
DBW2 gets latch 3 (250) and latch 5 (100) for 350.

Thus, the load carried by each of the DBW*n* processes differs, and performance suffers. If, however, there are 3 latches in each pool, then the DBW*n* processes have equal loads, and performance is optimized.

The different buffer pools have different rates of block replacement. Ordinarily, blocks are rarely modified in the `KEEP` pool and frequently modified in the `RECYCLE` pool. This means that you need to write out blocks more frequently from the `RECYCLE` pool than from the `KEEP` pool. As a result, owning 100 buffers from one pool is not the same as owning 100 buffers from the other pool. To be perfectly load balanced, each DBW*n* process should have the same number of LRU lists from each type of buffer pool.

A well-configured system might have 3 DBW*n* processes and 9 latches, with 3 latches in each buffer pool.

Figure 20–4   LRU Latches with Multiple Buffer Pools: Example 2



The `DEFAULT` buffer pool has 500 buffers for each LRU list. The `RECYCLE` buffer pool has 250 buffers for each LRU list. The `KEEP` buffer pool has 100 buffers for each LRU list.

DBW0 gets latch 1 (500) and latch 4 (250) and latch 7 (100) for 850.
DBW1 gets latch 2 (500) and latch 5 (250) and latch 8 (100) for 850.
DBW2 gets latch 3 (500) and latch 6 (250) and latch 9 (100) for 850.

Tuning Backup and Restore Operations

The primary goal of backup and restore tuning is to create an adequate flow of data between disk and storage device. Tuning backup and restore operations involve the following tasks:

- Locating the Source of Bottlenecks
- Using Recovery Manager
- Using Fixed Views to Monitor Bottlenecks
- Improving Backup Throughput

Locating the Source of Bottlenecks

Backups and restore operations have three distinct components:

- Read the input (disk or tape).
- Process data by validating blocks and copying them from the input to the output buffer.
- Write the output to tape or disk.

It is unlikely that all three of these perform at the same speed. Therefore, the slowest of these components is the bottleneck.

Types of I/O Oracle backup and restore uses two types of I/O: disk and tape. When performing a backup, the input files are read using disk I/O, and the output backup file is written using either disk or tape I/O. When performing restores, these roles reverse. Both disk and tape I/O can be synchronous or asynchronous; each is independent of the other.

Measuring Synchronous and Asynchronous I/O Rates When using synchronous I/O, you can easily determine how much time backup jobs require, because devices only perform one I/O task at a time. When using asynchronous I/O, it is more difficult to measure the bytes-per-second rate for the following reasons:

- Asynchronous processing implies that more than one task occurs at a time.
- Oracle I/O uses a polling, rather than an interrupt, mechanism to determine when each I/O request completes. Because the backup or restore process is not immediately notified of I/O completion by the operating system, you cannot determine the duration of each I/O.

Using Recovery Manager

Recovery Manager (RMAN) is an Oracle tool that allows you to back up, copy, restore, and recover datafiles, control files, and archived redo logs. You can invoke RMAN as a command-line utility from the operating system prompt or use the GUI-based Enterprise Manager Backup Manager.

RMAN automates many of the backup and recovery tasks that were formerly performed manually. For example, instead of requiring you to locate appropriate backups for each datafile, copy them to the correct place using operating system commands, and choose which archived logs to apply, RMAN manages all these tasks automatically.

RMAN provides several parameters that allow you to tune backup and recovery operations. These are discussed in the following sections.

Allocating Disk Buffers There are two different buffers: disk buffers and tape buffers. They can be different sizes. When RMAN backs up from disk, it allocates four disk buffers for each input datafile. You cannot alter the number of buffers that RMAN allocates.

The size of the disk buffers is controlled by the `DB_FILE_DIRECT_IO_COUNT` initialization parameter. This is the number of blocks per buffer. The default is 64.

The size of each buffer is equal to the product of the following initialization parameters:

```
DB_BLOCK_SIZE * DB_FILE_DIRECT_IO_COUNT
```

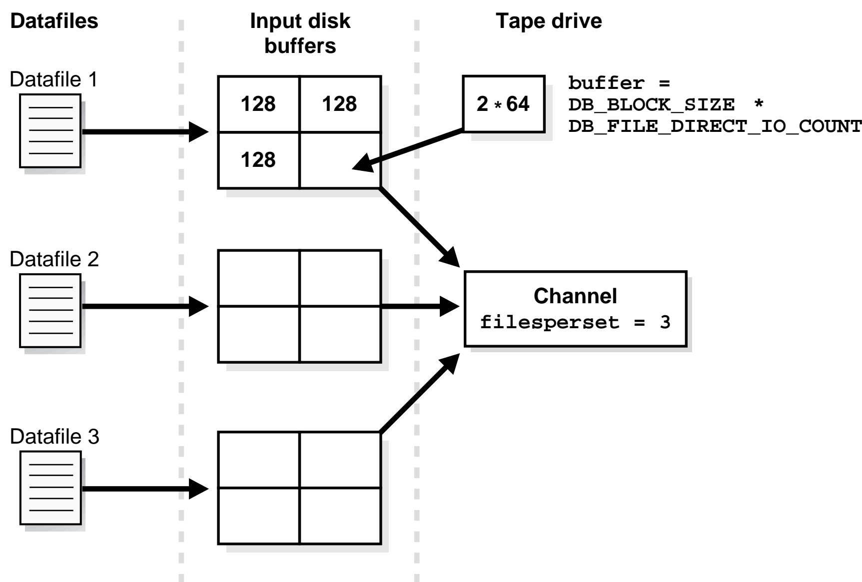
For example, if `DB_BLOCK_SIZE` = 2048 and `DB_FILE_DIRECT_IO_COUNT` =64, then each disk buffer is 128K. In this example, the total size of the buffers for each datafile is 128K * 4, or 512K. There are 4 buffers allocated for each datafile in the backup set.

If you want to know the total size of the buffers allocated in your backup, then multiply this total by the number of datafiles being accessed by the channel, and then multiply by the number of channels. You should also add a little extra to account for the control structures.

> **Note:** On some platforms, the most efficient I/O buffer size may be more than 128KB.

You can reduce the size of the buffers by lowering `DB_FILE_DIRECT_IO_COUNT`, but the number of buffers remains at 4 per file.

Figure 20–5 Disk Buffer Allocation



Allocating Tape Buffers Oracle allocates 4 buffers per channel for the tape writers (or reads if doing a restore). There are usually 64K each. Therefore, to size this, multiply by 4, and then multiply by the number of channels.

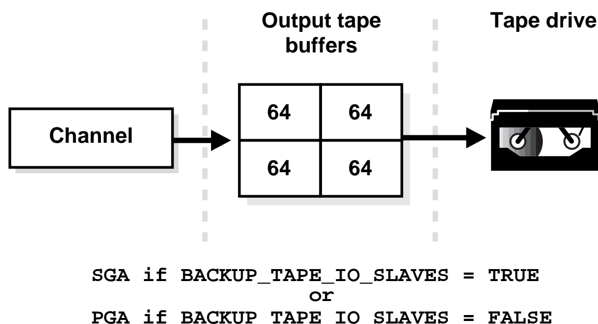
You can change the size of each tape buffer using the **parms** parameter of the `ALLOCATE CHANNEL` statement. Set **blksize** to the desired size of each buffer. For example:

```
allocate channel foo type 'sbt_tape' parms="blksize=16384"
```

RMAN allocates the tape buffers in the SGA or the PGA. If you set the initialization parameter `BACKUP_TAPE_IO_SLAVES` = `true`, then RMAN allocates them from the SGA. If you set the parameter to `FALSE`, then RMAN allocates the buffers in the PGA.

If you use I/O slaves, then you should use the `LARGE_POOL_SIZE` initialization parameter to set aside some SGA memory that is dedicated to holding these large memory allocations. Hence, the RMAN I/O buffers do not compete with the library cache for SGA memory.

Figure 20–6 Tape Buffer Allocation



Synchronous vs. Asynchronous I/O When RMAN reads or writes data, the action is either synchronous or asynchronous. When the I/O is synchronous, a server process can perform only one task at a time. When the I/O is asynchronous, a server process can begin one task, and other processes can perform other tasks while the initial process waits for the task to complete.

You can set initialization parameters that determine the type of I/O. If you set `BACKUP_TAPE_IO_SLAVES` to `true`, then the I/O is asynchronous. Otherwise, the I/O is synchronous.

Figure 20–7 illustrates *synchronous* I/O in a backup to tape. The following steps occur:

1. A server process writes blocks to a tape buffer.
2. The tape process writes data to tape. *While the tape process is writing, the server process must remain idle.*
3. The tape process returns a message to the server process stating that it has completed writing.
4. The server process can initiate a new task.

Figure 20–7 Synchronous I/O

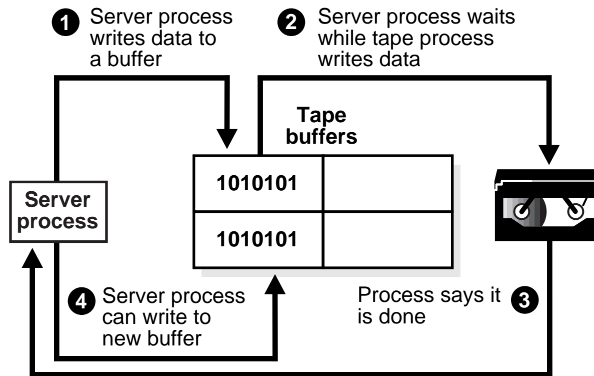
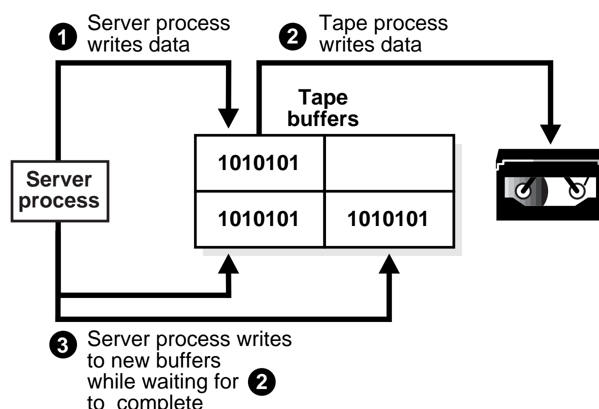


Figure 20–8 illustrates *asynchronous* I/O in a backup to tape. The following steps occur:

1. A server process writes blocks to a tape buffer.
2. The tape process writes data to tape. *While the tape process is writing, other server processes are free to perform tasks.*
3. Two spawned server processes write to tape buffers as the initial tape process writes to tape.

Figure 20–8 Asynchronous I/O



Allocating Channels When you allocate a channel, RMAN lets you set various channel limit parameters that apply to operations performed by the allocated server session. You can use these parameters to do the following:

- Force RMAN to create multiple backup pieces.
- Prevent RMAN from consuming too much disk bandwidth.
- Prevent RMAN from opening too many input files at once.

You can specify the following parameters:

Parameter	Description
kbytes	Specifies the maximum size of a backup piece created on a channel. Use this parameter to force RMAN to create multiple backup pieces in a backup set. RMAN creates each backup piece with a size no larger than the value specified in the parameter.
readrate	Specifies the maximum number of buffers per second read from each input datafile. You can use this parameter to prevent RMAN from consuming too much bandwidth during backups. For example, set **readrate** to 12. If each input buffer is 128K, then the read rate for each input datafile is 12 * 128 or about 1.5 Mb per second. If each SCI drive delivers 3 Mb per second, then RMAN leaves some disk bandwidth available to the online system.

Parameter	Description
maxopenfiles	Determines the maximum number of input files that a backup or copy can have open at a given time. Set this parameter to prevent RMAN from attempting to open a number of files greater than the upper limit for your operating system.

See Also: For syntax information on the `ALLOCATE CHANNEL` statement, see *Oracle8i Recovery Manager User's Guide and Reference*.

Allocating Input Files The `BACKUP` statement lets you set parameters that influence how RMAN selects files for input into backup sets. You may need to set these parameter to do the following:

- Prevent RMAN from write a single backup set to multiple volumes.
- Prevent RMAN from reading from too many disks at once.
- Keep a tape drive streaming during a backup.

You can specify the following parameters:

Parameter	Description
filesperset	Specifies the maximum number of files to place in a backup set. RMAN divides the number of input files by the number of channels to calculate the number of files per backup set. Use this parameter to force RMAN to create multiple backup sets. For example, if you have 50 input datafiles and 2 channels, you can set **filesperset** = 5 to create 10 backup sets. This action can prevent you from splitting a backup set among multiple tapes.
diskratio	Specifies the number of drives to include in the backup. For example, assume your datafiles are located on 5 disks, the disks supply data at 10 bytes/second, and the tape drive requires 20 bytes/second to set keep streaming. If you set **diskratio** to 2, then RMAN reads from 2 drives at a time, thereby spreading the backup load.

> **Note:** The number of datafiles accessed by a channel can be controlled by setting **filesperset** on the `BACKUP` statement or by entering a `SET LIMIT CHANNEL` ... **maxopenfiles**=`n` statement before the `BACKUP` statement.

> **See Also:** For syntax information on the `BACKUP` statement, see *Oracle8i Recovery Manager User's Guide and Reference.*

Using Incremental Backups An incremental backup is an RMAN backup in which only modified blocks are backed up. Incremental backups are not necessarily faster than full backups, because Oracle still reads the entire datafile to take an incremental backup. If tape drives are locally attached, then incremental backups could be faster. You must consider how much bandwidth there is for reading the disks compared to the bandwidth for writing to the tapes. If tape bandwidth is limited compared to disk, then incremental backups could help.

In an incremental backup, if only a few blocks have changed, then you need to input many buffers from the datafile before you accumulate enough blocks to fill a buffer and write to tape. So, it is possible that the tape drive is not *streaming*. Streaming means that the tape drive is 100% busy. If a tape drive is not kept streaming, then it becomes inefficient, because after each write, the tape drive must stop and rewind a little bit.

If you use a large filesperset in the **filesperset** parameter, then you can scan many datafiles in parallel, the output buffers for the tape drive are filled quickly, and you can write them frequently enough to keep the tape drive streaming.

For an incremental backup, **filesperset**=50 might be a good number. But, for a full or incremental level=0 backup, **filesperset** should be smaller, such as 4 or 8.

RMAN Performance Tips To get the best performance for a backup, follow these suggestions:

1. Do not set the **readrate** parameter. This is intended to slow down a backup, so that you can run it in the background during production hours.
2. Set the `BACKUP_TAPE_IO_SLAVES` initialization parameter to `TRUE`.
3. If you set `BACKUP_TAPE_IO_SLAVES` to `TRUE`, then the tape buffers are allocated from the SGA. Therefore, allocate a `LARGE_POOL` for them. You can control the buffer size with the **parms** clause on the **allocate channel** statement:

4. Increase the size of the disk reads by setting the DB_FILE_DIRECT_IO_COUNT initialization parameter. Use the **maxopenfiles** parameter to control how many datafiles are opened simultaneously on each channel.
5. If your datafiles are in a UNIX file system, then try setting BACKUP_DISK_IO_SLAVES to 4. This spawns processes to read datafiles in parallel, simulating asynchronous I/O. In this case, the datafile buffers are allocated from the SGA. The default value for this parameter is 0, meaning I/O slaves are not used at all, and the buffers come from the process local memory, not the SGA.
6. For incremental backups, use a higher **filesperset** than for full backups. Set it high enough so that the tape drive is kept streaming. **Filesperset** should be less than or equal to **maxopenfiles**. Try making FILESPERSET=MAXOPENFILES. Use a value of 10 to start, and raise this if the tape drive does not stream. BACKUP_DISK_IO_SLAVES may be necessary.

Using Fixed Views to Monitor Bottlenecks

Use the V$BACKUP_SYNC_IO and V$BACKUP_ASYNC_IO views to determine the source of backup or restore bottlenecks and to determine the progress of backup jobs.

V$BACKUP_SYNC_IO contains rows when the I/O is synchronous to the process (or *thread* on some platforms) performing the backup. V$BACKUP_ASYNC_IO contains rows when the I/O is asynchronous. Asynchronous I/O is obtained either with I/O processes or because it is supported by the underlying operating system.

> **See Also:** For more information about these views, see the *Oracle8i Reference.*

Identifying Bottlenecks with Synchronous I/O With synchronous I/O, it is difficult to identify specific bottlenecks, because all synchronous I/O is a bottleneck to the process. The only way to tune synchronous I/O is to compare the bytes-per-second rate with the device's maximum throughput rate. If the bytes-per-second rate is lower than that device specifies, then consider tuning that part of the backup/restore process. Use the DISCRETE_BYTES_PER_SECOND column in the V$BACKUP_SYNC_IO view to see the I/O rate.

Identifying Bottlenecks with Asynchronous I/O If the combination of LONG_WAITS and SHORT_WAITS is a significant fraction of IO_COUNT, then the file indicated in V$BACKUP_SYNCH_IO and V$BACKUP_ASYNCH_IO is probably a bottleneck. Some platforms' implementation of asynchronous I/O can cause the caller to wait for I/O completion when performing a non-blocking poll for I/O. Because this behavior

can vary among platforms, the V$BACKUP_ASYNC_IO view shows the total time for both "short" and "long" waits.

Long waits are the number of times the backup/restore process told the operating system to wait until an I/O was complete. *Short* waits are the number of times the backup/restore process made an operating system call to poll for I/O completion in a non-blocking mode. Both types of waits the operating system should respond immediately.

If the SHORT_WAIT_TIME_TOTAL column is equal to or greater than the LONG_WAIT_TIME_TOTAL column, then your platform probably blocks for I/O completion when performing *non-blocking* I/O polling. In this case, the SHORT_WAIT_TIME_TOTAL represents real I/O time for this file. If the SHORT_WAIT_TIME_TOTAL is low compared to the total time for this file, then the delay is most likely caused by other factors, such as process swapping. If possible, tune your operating system so that the I/O wait time appears up in the LONG_WAIT_TIME_TOTAL column.

Columns Common to V$BACKUP_SYNC_IO and V$BACKUP_ASYNC_IO Table 20–17 lists columns and their descriptions that are common to the V$BACKUP_SYNC_IO and V$BACKUP_ASYNC_IO views.

Table 20–17 Common Columns of V$BACKUP_SYNC_IO and V$BACKUP_ASYNC_IO

Column	Description
SID	Oracle SID of the session doing the backup or restore.
SERIAL	Usage counter for the SID doing the backup or restore.
USE_COUNT	A counter you can use to identify rows from different backup sets. Each time a new set of rows is created in V$BACKUP_SYNC_IO or V$BACKUP_ASYNC_IO, they have a USE_COUNT that is greater than the previous rows. The USE_COUNT is the same for all rows used by each backup or restore operation.
DEVICE_TYPE	Device type where the file is located (typically DISK or SBT_TAPE).
TYPE	INPUT: The file(s) are being read. OUTPUT: The file(s) are being written. AGGREGATE: This row represents the total I/O counts for all DISK files involved in the backup or restore.

Table 20–17 Common Columns of V$BACKUP_SYNC_IO and V$BACKUP_ASYNC_IO

Column	Description
STATUS	NOT STARTED: This file has not been opened yet. IN PROGRESS: This file is currently being read or written. FINISHED: Processing for this file is complete.
FILENAME	The name of the backup file being read or written.
SET_COUNT	The SET_COUNT of the backup set being read or written.
SET_STAMP	The SET_STAMP of the backup set being read or written.
BUFFER_SIZE	Size of the buffers being used to read/write this file in bytes.
BUFFER_COUNT	The number of buffers being used to read/write this file.
TOTAL_BYTES	The total number of bytes to be read or written for this file if known. If not known, this column is null.
OPEN_TIME	Time this file was opened. If TYPE = 'AGGREGATE', then this is the time that the first file in the aggregate was opened.
CLOSE_TIME	Time this file was closed. If TYPE = 'AGGREGATE', then this is the time that the last file in the aggregate was closed.
ELAPSED_TIME	The length of time expressed in 100ths of seconds that the file was open.
MAXOPENFILES	The number of concurrently open DISK files. This value is only present in rows where TYPE = 'AGGREGATE'.
BYTES	The number of bytes read or written so far.
EFFECTIVE_BYTES_PER_SECOND	The I/O rate achieved with this device during the backup. It is the number of bytes read or written divided by the elapsed time. This value is only meaningful for the component of the backup system causing a bottleneck. If this component is not causing a bottleneck, then the speed measured by this column actually reflects the speed of some other, slower, component of the system.
IO_COUNT	The number of I/Os performed to this file. Each request is to read or write one buffer, of size BUFFER_SIZE.

Columns Specific to V$BACKUP_SYNC_IO Table 20–18 lists columns specific to the V$BACKUP_SYNC_IO view.

Table 20–18 Columns Specific to V$BACKUP_SYNC_IO

Column	Description
IO_TIME_TOTAL	The total time required to perform I/O for this file expressed in 100ths of seconds.
IO_TIME_MAX	The maximum time taken for a single I/O request.
DISCRETE_BYTES_PER_SECOND	The average transfer rate for this file. This is based on measurements taken at the start and end of each individual I/O request. This value should reflect the real speed of this device.

Columns Specific to V$BACKUP_ASYNC_IO Table 20–19 lists columns specific to the V$BACKUP_ASYNC_IO view.

Table 20–19 Columns Specific to V$BACKUP_ASYNC_IO

Column	Description
READY	The number of asynchronous requests for which a buffer was immediately ready for use.
SHORT_WAITS	The number of times a buffer was not immediately available, but then a buffer became available after doing a non-blocking poll for I/O completion. The reason non-blocking waits are timed is because some implementations of *asynchronous I/O* may wait for an I/O to complete even when the request is supposed to be non-blocking.
SHORT_WAIT_TIME_TOTAL	The total time expressed in 100ths of seconds, taken by non-blocking polls for I/O completion.
SHORT_WAIT_TIME_MAX	The maximum time taken for a non-blocking poll for I/O completion, in 100ths of seconds.
LONG_WAITS	The number of times a buffer was not immediately available, and only became available after issuing a blocking wait for an I/O to complete.
LONG_WAIT_TIME_TOTAL	The total time expressed in 100ths of seconds taken by blocking waits for I/O completion.
LONG_WAIT_TIME_MAX	The maximum time taken for a blocking wait for I/O completion expressed in 100ths of seconds.

Improving Backup Throughput

In optimally tuned backups, tape components should create the only bottleneck. You should keep the tape and its device *streaming*, or constantly rotating. If the tape is not streaming, then the data flow to the tape may be inadequate.

This section contains the following topics to maintain streaming by improving backup throughput:

- Understanding Factors Affecting Data Transfer Rates
- Determining If Tape is Streaming for Synchronous I/O
- Determining If Tape is Streaming for Asynchronous I/O
- Increasing Throughput to Enable Tape Streaming
- Spreading I/O Across Multiple Disks
- Backing Up Empty Files or Files with Few Changes
- Backing Up Full Files

Understanding Factors Affecting Data Transfer Rates The rate at which the host sends data to keep the tape streaming depends on the following factors:

- The raw capacity of the tape device.
- Compression.

Tape device raw capacity is the *smallest* amount of data required to keep the tape streaming.

Compression is implemented either in the tape hardware or by the media management software. If you do not use compression, then the raw capacity of the tape device keeps it streaming. If you use compression, then the amount of data that must be sent to stream the tape is the raw device capacity multiplied by the compression factor. The compression factor varies for different types of data.

Determining If Tape is Streaming for Synchronous I/O To determine whether your tape is streaming when the I/O is synchronous, query the `EFFECTIVE_BYTES_PER_SECOND` column in the `V$BACKUP_SYNC_IO` view.

Table 20–20 V$BACKUP_SYNC_IO View

If EFFECTIVE_BYTES_PER_SECOND is:	Then:
Less than the raw capacity of the hardware	The tape is not streaming.
More than the raw capacity of the hardware	The tape may be streaming, depending on the compression ratio of the data.

Determining If Tape is Streaming for Asynchronous I/O If the I/O is asynchronous, then the tape is streaming if the combination of `LONG_WAITS` and `SHORT_WAITS` is a significant fraction of I/O count. Place more importance on `SHORT_WAITS` if the time indicated in the `SHORT_WAIT_TIME_TOTAL` column is equal to or greater than the `LONG_WAIT_TIME_TOTAL` column.

Increasing Throughput to Enable Tape Streaming If the tape is not streaming, then the basic strategy is to supply more bytes-per-second to the tape. Modify this strategy depending on the how many blocks Oracle must read from disk and how many disks Oracle must access.

Spreading I/O Across Multiple Disks Using the `DISKRATIO` parameter of the `BACKUP` statement to distribute backup I/O across multiple volumes, specify how many disk drives RMAN uses to distribute file reads when backing up multiple concurrent datafiles. For example, assume that your system uses 10 disks. The disks supply data at 10 byes per second, and the tape drive requires 50 bytes per second to keep streaming. In this case, set `DISKRATIO` to 5 to spread the backup load onto 5 disks.

When setting `DISKRATIO`, spread the I/O over only as many disks as needed to keep the tape streaming: any more can increase the time it takes to recover a single file and provides no performance benefit. If you do not specify `DISKRATIO`, but you do specify `FILESPERSET`, then `DISKRATIO` defaults to `FILESPERSET`. If neither is specified, then `DISKRATIO` defaults to 4.

Backing Up Empty Files or Files with Few Changes When performing a full backup of files that are largely empty, or when performing an incremental backup when few blocks have changed, you may not be able to supply data to the tape fast enough to keep it streaming.

In this case, you can achieve optimal performance by the following:

- Use the highest possible value for the `MAXOPENFILES` parameter of the Recovery Manager `SET LIMIT CHANNEL` statement.

> **See Also:** For more information about the RMAN SET statement, see *Oracle8i Backup and Recovery Guide*.

- Use asynchronous disk I/O. This takes advantage of asynchronous read-ahead that fills input buffers from one file while processing data from others.

Backing Up Full Files When you perform a full backup of files that are mostly full and the tape is not streaming, you can improve performance in several ways, as shown in Table 20–21:

Table 20–21 Throughput Performance Improvement Methods

Method	Consequence
Set DBWR_IO_SLAVES = TRUE	Allocates additional processes for each disk channel, and these processes simulate asynchronous I/O. Try setting this to 3 or 4, and set the LARGE_POOL_SIZE parameter accordingly.
Set BACKUP_TAPE_IO_SLAVES = TRUE	Simulates asynchronous tape I/O by spawning multiple processes to divide the work for the backup or restore operation. If you do not set this parameter, then all I/O to the tape layer is synchronous, which means that no other work can be done until the tape is done writing. BACKUP_TAPE_IO_SLAVES requires that the buffers for the respective disk or tape I/O be allocated from the shared memory (SGA), so that they can be shared between two processes. Therefore, you should allocate a large enough SGA. If you set this parameter, then also set LARGE_POOL_SIZE.
Set LARGE_POOL_SIZE	When attempting to get shared buffers for I/O slaves, Oracle does the following: ■ If LARGE_POOL_SIZE is set, then Oracle attempts to get memory from the large pool. If this value is not large enough, then Oracle does not try to get buffers from the shared pool. ■ If LARGE_POOL_SIZE is not set, then Oracle attempts to get memory from the shared pool. ■ If Oracle cannot get enough memory, then it obtains I/O buffer memory from local process memory and writes a message to the alert.log file indicating that synchronous I/O is used for this backup.

Table 20–21 Throughput Performance Improvement Methods

Method	Consequence
Increase DB_FILE_DIRECT_IO_COUNT	Causes RMAN to use larger buffers for disk I/O. The default buffer size used for backup and restore disk I/O is DB_FILE_DIRECT_IO_COUNT * DB_BLOCK_SIZE. The default value for DB_FILE_DIRECT_IO_COUNT is 64, so if DB_BLOCK_SIZE is 2048, then the buffer size is 128KB. On some platforms, the most efficient I/O buffer size may be more than 128KB. You can increase the DB_FILE_DIRECT_IO_COUNT, but the number of buffers per file stays fixed at 4.

Make sure the RMAN parameters MAXOPENFILES and FILESPERSET are not too low	Increases the number of files that RMAN can process at one time. Using default buffer sizes, each concurrently open file uses 512KB of process memory (or SGA large pool memory, if I/O processes are used) for buffers. The number of concurrent files should be just enough to keep the tape streaming.
	
	You must derive the correct number by trial and error, because unused block compression greatly affects the amount of disk data that is sent to the tape drive. If your tape drives are slower than your disk drives, then a value of 1 for MAXOPENFILES should be sufficient.
	
	If you want a high FILESPERSET, but you want to limit the memory allocated for buffers, then use SETLIMIT. For example:
	
	`run{`
	`allocate channel foo type disk;`
	`SETLIMIT CHANNEL foo maxopenfiles=4`
	`backup database...}`
	
	SETLIMIT limits the channels to have at most 4 open files. So, you could allocate 16 buffers for the input files and 4 buffers for the backup set. The default is 10, which may be high for your system.

READRATE	The READRATE parameter specifies units of buffers per second. For example, if your buffers are 128K and READRATE is 12, then RMAN is limited to reading 12*128K bytes per second from each datafile going into a backup set. You should test to find a value that improves performance of your queries while still letting RMAN complete the backup in a reasonable amount of time.

Table 20–21 Throughput Performance Improvement Methods

Method	Consequence

Increase the number of channels	Increases parallelism. Each channel must write to different filesystems, which should be on different disks. One channel per tape drive ensures that during restore, the files are read with the same sequence and timing as when they were created.
	
	You must specify the `FORMAT` parameter on the `ALLOCATE CHANNEL` statement. For example:
	
	`run{`
	`allocate channel foo1 type disk format '/filesys1/oracle_backups/%d/%u_%p';`
	`allocate channel foo2 type disk format '/filesys2/oracle_backups/%d/%u_%p';`
	`...}`

Configuring the Large Pool

You can optionally configure the large pool so that Oracle has a separate pool from which it can request large memory allocations. This prevents competition with other subsystems for the same memory.

As Oracle allocates more shared pool memory for the multi-threaded server session memory, the amount of shared pool memory available for the shared SQL cache decreases. If you allocate session memory from another area of shared memory, then Oracle can use the shared pool primarily for caching shared SQL and not incur the performance overhead from shrinking the shared SQL cache.

For I/O server processes and backup and restore operations, Oracle allocates buffers that are a few hundred kilobytes in size. Although the shared pool may be unable to satisfy this request, the large pool is able to do so. The large pool does not have an LRU list; Oracle does not attempt to age memory out of the large pool.

Use the `LARGE_POOL_SIZE` parameter to configure the large pool. To see in which pool (shared pool or large pool) the memory for an object resides, see the column `POOL` in `V$SGASTAT`.

> **See Also:** For more information about the large pool, see *Oracle8i Concepts*. For complete information about initialization parameters, see *Oracle8i Reference.*

21

Tuning Resource Contention

Contention occurs when multiple processes try to access the same resource simultaneously. Some processes must then wait for access to various database structures.

This chapter contains the following sections:

- Understanding Contention Issues
- Detecting Contention Problems
- Solving Contention Problems

Understanding Contention Issues

Symptoms of resource contention problems can be found in V$SYSTEM_EVENT. This view reveals various system problems that may be impacting performance, problems such as latch contention, buffer contention, and I/O contention. It is important to remember that these are only *symptoms* of problems—not the actual causes.

For example, by looking at V$SYSTEM_EVENT you might notice lots of buffer-busy waits. It may be that many processes are inserting into the same block and must wait for each other before they can insert. The solution might be to introduce free lists for the object in question.

Buffer busy waits may also have caused some latch free waits. Because most of these waits were caused by misses on the cache buffer hash chain latch, this was also a side effect of trying to insert into the same block. Rather than increasing SPINCOUNT to reduce the latch free waits (a symptom), you should change the object to allow for multiple processes to insert into free blocks. This approach effectively reduces contention.

Detecting Contention Problems

The V$RESOURCE_LIMIT view provides information about current and maximum global resource utilization for some system resources. This information enables you to make better decisions when choosing values for resource limit-controlling parameters.

If the system has idle time, then start your investigation by checking V$SYSTEM_EVENT. Examine the events with the highest average wait time, then take appropriate action on each. For example, if you find a high number of latch free waits, then look in V$LATCH to see which latch is the problem.

For excessive buffer busy waits, look in V$WAITSTAT to see which block type has the highest wait count and the highest wait time. Look in V$SESSION_WAIT for cache buffer waits so you can decode the file and block number of an object.

The rest of this chapter describes common contention problems. Remember that the different forms of contention are symptoms which can be fixed by making changes in one of two places:

- Changes in the application.
- Changes in Oracle.

Sometimes you have no alternative but to change the application in order to overcome performance constraints.

Solving Contention Problems

The rest of this chapter examines various kinds of contention and explains how to resolve problems. Contention may be for rollback segments, multi-threaded servers, parallel execution servers, redo log buffer latches, LRU latch, or for free lists.

Reducing Contention for Rollback Segments

This section discusses how to reduce contention for rollback segments. The following issues are explained:

- Identifying Rollback Segment Contention
- Creating Rollback Segments

Identifying Rollback Segment Contention

Contention for rollback segments is reflected by contention for buffers that contain rollback segment blocks. You can determine whether contention for rollback segments is adversely affecting performance by checking the dynamic performance table V$WAITSTAT.

V$WAITSTAT contains statistics that reflect block contention. By default, this table is available only to the user SYS and to other users who have SELECT ANY TABLE system privilege, such as SYSTEM. These statistics reflect contention for different classes of blocks:

SYSTEM UNDO HEADER	The number of waits for buffers containing header blocks of the SYSTEM rollback segment.
SYSTEM UNDO BLOCK	The number of waits for buffers containing blocks of the SYSTEM rollback segment other than header blocks.
UNDO HEADER	The number of waits for buffers containing header blocks of rollback segments other than the SYSTEM rollback segment.
UNDO BLOCK	The number of waits for buffers containing blocks other than header blocks of rollback segments other than the SYSTEM rollback segment.

Use the following query to monitor these statistics over a period of time while your application is running:

```
SELECT CLASS, COUNT
    FROM V$WAITSTAT
    WHERE CLASS IN ('SYSTEM UNDO HEADER', 'SYSTEM UNDO BLOCK',
        'UNDO HEADER', 'UNDO BLOCK');
```

The result of this query might look like this:

```
CLASS                   COUNT
------------------ ----------
SYSTEM UNDO HEADER       2089
SYSTEM UNDO BLOCK         633
UNDO HEADER              1235
UNDO BLOCK                942
```

Compare the number of waits for each class of block with the total number of requests for data over the same period of time. You can monitor the total number of requests for data over a period of time with this query:

```
SELECT SUM(VALUE)
    FROM V$SYSSTAT
    WHERE NAME IN ('DB BLOCK GETS', 'CONSISTENT GETS');
```

The output of this query might look like this:

```
SUM(VALUE)
----------
929530
```

The information in `V$SYSSTAT` can also be obtained through SNMP.

If the number of waits for any class of block exceeds 1% of the total number of requests, then consider creating more rollback segments to reduce contention.

Creating Rollback Segments

To reduce contention for buffers containing rollback segment blocks, create more rollback segments. Table 21–1 shows some general guidelines for choosing how many rollback segments to allocate based on the number of concurrent transactions on your database. These guidelines are appropriate for most application mixes.

Table 21–1 Choosing the Number of Rollback Segments

Number of Current Transactions (*n*)	Number of Rollback Segments Recommended
n < 16	4
16 <= n < 32	8
32 <= n	n/4

Reducing Contention for Multi-Threaded Servers

Performance of certain database features may degrade slightly when MTS is used. These features include `BFILE`s, parallel execution, inter-node parallel execution, and hash joins. This is because these features may prevent a session from migrating to another shared server while they are active.

A session may remain non-migratable after a request from the client has been processed. Use of the above mentioned features may make sessions non-migratable, because the features have not stored all the user state information in the UGA, but have left some of the state in the PGA. As a result, if different shared servers process requests from the client, then the part of the user state stored in the PGA is inaccessible. To avoid this, individual shared servers often need to remain bound to a user session. This makes the session non-migratable among shared servers.

When using these features, you may need to configure more shared servers. This is because some servers may be bound to sessions for an excessive amount of time.

This section discusses how to reduce contention for processes used by Oracle's multi-threaded server (MTS) architecture:

- Identifying Contention Using the Dispatcher-Specific Views
- Reducing Contention for Dispatcher Processes
- Reducing Contention for Shared Servers
- Determining the Optimal Number of Dispatchers and Shared Servers

Identifying Contention Using the Dispatcher-Specific Views

The following views provide dispatcher performance statistics:

- `V$DISPATCHER`
- `V$DISPATCHER_RATE`

V$DISPATCHER provides general information about dispatcher processes. V$DISPATCHER_RATE view provides dispatcher processing statistics.

> **See Also:** For detailed information about these views, see the *Oracle8i Reference*.

Analyzing V$DISPATCHER_RATE Statistics The V$DISPATCHER_RATE view contains current, average, and maximum dispatcher statistics for several categories. Statistics with the prefix "CUR_" are statistics for the current session. Statistics with the prefix "AVG_" are the average values for the statistics since the collection period began. Statistics with "MAX_" prefixes are the maximum values for these categories since statistics collection began.

To assess dispatcher performance, query the V$DISPATCHER_RATE view and compare the current values with the maximums. If your present system throughput provides adequate response time and current values from this view are near the average and below the maximum, then you likely have an optimally-tuned MTS environment.

If the current and average rates are significantly below the maximums, then consider reducing the number of dispatchers. Conversely, if current and average rates are close to the maximums, then you may need to add more dispatchers. A good rule-of-thumb is to examine V$DISPATCHER_RATE statistics during both light and heavy system use periods. After identifying your MTS load patterns, adjust your parameters accordingly.

If needed, you can also mimic processing loads by running system stress-tests and periodically polling the V$DISPATCHER_RATE statistics. Proper interpretation of these statistics varies from platform to platform. Different types of applications also can cause significant variations on the statistical values recorded in V$DISPATCHER_RATE.

Reducing Contention for Dispatcher Processes

This section discusses how to identify contention for dispatcher processes, how to add dispatcher processes, and how to enable connection pooling.

Identifying Contention for Dispatcher Processes Contention for dispatcher processes is indicated by either of these symptoms:

- Excessive busy rates for existing dispatcher processes.
- Steady increases in waiting times for responses in the response queues of dispatcher processes.

Examining Busy Rates for Dispatcher Processes V$DISPATCHER contains statistics reflecting the activity of dispatcher processes. By default, this view is available only to the user SYS and to other users who have SELECT ANY TABLE system privilege, such as SYSTEM. These columns reflect busy rates for dispatcher processes:

IDLE	Displays the idle time for the dispatcher process in hundredths of a second.
BUSY	Displays the busy time for the dispatcher process in hundredths of a second.

If the database is only in use 8 hours per day, then statistics need to be normalized by the effective work times. You cannot simply look at statistics from the time the instance started. Instead, record statistics during peak workloads. If the dispatcher processes for a specific protocol are busy for more than 50% of the peak workload period, then by adding dispatcher processes, you may improve performance for users connected to Oracle using that protocol.

Examining Wait Times for Dispatcher Process Response Queues V$QUEUE contains statistics reflecting the response queue activity for dispatcher processes. By default, this table is available only to the user SYS and to other users who have SELECT ANY TABLE system privilege, such as SYSTEM. These columns show wait times for responses in the queue:

WAIT	The total waiting time, in hundredths of a second, for all responses that have ever been in the queue.
TOTALQ	The total number of responses that have ever been in the queue.

Use the following query to monitor these statistics occasionally while your application is running:

```
SELECT CONF_INDX "INDEX",
    DECODE( SUM(TOTALQ), 0, 'NO RESPONSES',
        SUM(WAIT)/SUM(TOTALQ) || ' HUNDREDTHS OF SECONDS')
    "AVERAGE WAIT TIME PER RESPONSE"
    FROM V$QUEUE Q, V$DISPATCHER D
    WHERE Q.TYPE = 'DISPATCHER'
        AND Q.PADDR = D.PADDR
    GROUP BY CONF_INDX;
```

This query returns the average time, in hundredths of a second, that a response waits in each response queue for a dispatcher process to route it to a user process. This query uses the V$DISPATCHER table to group the rows of the V$QUEUE table

by MTS_DISPATCHERS parameter value index. The query also uses the DECODE syntax to recognize those protocols for which there have been no responses in the queue. The result of this query might look like this:

```
INDEX     AVERAGE WAIT TIME PER RESPONSE
--------  ------------------------------
0         .1739130 HUNDREDTHS OF SECONDS
1         NO RESPONSES
```

From this result, you can tell that a response in the queue for the first MTS_DISPATCHERS value's dispatchers waits an average of 0.17 hundredths of a second, and that there have been no responses in the queue for the second MTS_DISPATCHERS value's dispatchers.

If the average wait time for a specific MTS_DISPATCHERS value continues to increase steadily as your application runs, then by adding dispatchers, you may be able to improve performance of those user processes connected to Oracle using that group of dispatchers.

Adding Dispatcher Processes Add dispatcher processes while Oracle is running by using the SET option of the ALTER SYSTEM statement to increase the value for the MTS_DISPATCHERS parameter.

The total number of dispatcher processes is limited by the value of the initialization parameter MTS_MAX_DISPATCHERS. You may need to increase this value before adding dispatcher processes. The default value of this parameter is 5 and the maximum value varies depending on your operating system.

> **See Also:** For more information on adding dispatcher processes, see *Oracle8i Administrator's Guide* and *Net8 Administrator's Guide.*

Enabling Connection Pooling When system load increases and dispatcher throughput is maximized, it is not necessarily a good idea to immediately add more dispatchers. Instead, consider configuring the dispatcher to support more users with connection pooling.

MTS_DISPATCHERS lets you enable various attributes for each dispatcher. Oracle supports a name-value syntax to let you specify attributes in a position-independent, case-insensitive manner. For example:

```
MTS_DISPATCHERS = "(PROTOCOL=TCP)(POOL=ON)(TICK=1)"
```

The optional attribute POOL is used to enable the Net8 connection pooling feature. TICK is the size of a network TICK in seconds. The TICK - default is 15 seconds.

> **See Also:** For more information about the MTS_DISPATCHERS parameter and its options, see the *Oracle8i SQL Reference* and the *Net8 Administrator's Guide*.

Enabling Connection Concentration Multiplexing is used by a connection manager process to establish and maintain connections from multiple users to individual dispatchers. For example, several user processes may connect to one dispatcher by way of a single connection manager process.

The connection manager manages communication from users to the dispatcher by way of the single connection. At any one time, zero, one, or a few users may need the connection, while other user processes linked to the dispatcher by way of the connection manager process are idle. In this way, multiplexing is beneficial as it maximizes use of user-to-dispatcher process connections.

Multiplexing is also useful for multiplexing database link connections between dispatchers. The limit on the number of connections for each dispatcher is platform dependent. For example:

```
MTS_DISPATCHERS="(PROTOCOL=TCP)(MULTIPLEX=ON)"
```

Reducing Contention for Shared Servers

This section discusses how to identify contention for shared servers and how to increase the maximum number of shared servers.

Identifying Contention for Shared Servers Steadily increasing wait times in the requests queue indicate contention for shared servers. To examine wait time data, use the dynamic performance view V$QUEUE. This view contains statistics showing request queue activity for shared servers. By default, this view is available only to the user SYS and to other users with SELECT ANY TABLE system privilege, such as SYSTEM. These columns show wait times for requests in the queue:

WAIT	Displays the total waiting time, in hundredths of a second, for all requests that have ever been in the queue.
TOTALQ	Displays the total number of requests that have ever been in the queue.

Monitor these statistics occasionally while your application is running by issuing the following SQL statement:

```
SELECT DECODE(TOTALQ, 0, 'No Requests',
    WAIT/TOTALQ || ' HUNDREDTHS OF SECONDS')
"AVERAGE WAIT TIME PER REQUESTS"
```

```
FROM V$QUEUE
WHERE TYPE = 'COMMON';
```

This query returns the results of a calculation that shows the following:

```
AVERAGE WAIT TIME PER REQUEST
-----------------------------
.090909 HUNDREDTHS OF SECONDS
```

From the result, you can tell that a request waits an average of 0.09 hundredths of a second in the queue before processing.

You can also determine how many shared servers are currently running by issuing this query:

```
SELECT COUNT(*) "Shared Server Processes"
    FROM V$SHARED_SERVER
    WHERE STATUS != 'QUIT';
```

The result of this query might look like this:

```
SHARED SERVER PROCESSES
-----------------------
10
```

If you detect resource contention with MTS, then first make sure that this is not a memory contention issue by examining the shared pool and the large pool. If performance remains poor, then you may want to create more resources to reduce shared server process contention. Do this by modifying the optional server process parameters as explained under the following headings.

Setting and Modifying MTS Processes This section explains how to set optional parameters affecting processes for the multi-threaded server architecture. This section also explains how and when to modify these parameters to tune performance.

The static initialization parameters discussed in this section are:

- MTS_MAX_DISPATCHERS
- MTS_MAX_SERVERS

This section also describes the initialization/session parameters:

- MTS_DISPATCHERS
- MTS_SERVERS

Values for the initialization parameters MTS_MAX_DISPATCHERS and MTS_MAX_SERVERS define upper limits for the number of dispatchers and servers running on an instance. These parameters are static and cannot be changed after your database is running. You can create as many dispatcher and server processes as you need, but the total number of processes cannot exceed the host operating system's limit for the number of running processes.

> **Note:** Setting MTS_MAX_DISPATCHERS sets the limit on the number of dispatchers for all MTS_DISPATCHERS' dispatcher values.

You can also define starting values for the number of dispatchers and servers by setting the MTS_DISPATCHERS parameter's DISPATCHER attribute and the MTS_SERVERS parameter. After system startup, you can dynamically re-set values for these parameters to change the number of dispatchers and servers using the SET option of the ALTER SYSTEM statement. If you enter values for these parameters in excess of limits set by the static parameters, then Oracle uses the static parameter values.

The default value of MTS_MAX_SERVERS is dependent on the value of MTS_SERVERS. If MTS_SERVERS is less than or equal to 10, then MTS_MAX_SERVERS defaults to 20. If MTS_SERVERS is greater than 10, then MTS_MAX_SERVERS defaults to 2 times the value of MTS_SERVERS.

Self-adjusting MTS Architecture Features When the database starts, MTS_SERVERS is the number of shared servers created. Oracle will not allow the number of shared servers to fall below this minimum. During processing, Oracle automatically adds shared servers up to the limit defined by MTS_MAX_SERVERS if Oracle perceives that the load based on the activity of the requests on the common queue warrant additional shared servers. Therefore, you are unlikely to improve performance by explicitly adding shared servers. However, you may need to adjust your system to accommodate certain resource issues.

If the number of shared server processes has reached the limit set by the initialization parameter MTS_MAX_SERVERS and the average wait time in the request queue is still unacceptable, then you might improve performance by increasing the MTS_MAX_SERVERS value.

If resource demands exceed expectations, then you can either allow Oracle to automatically add shared server processes or you can add shared processes by altering the value for MTS_SERVERS. You can change the value of this parameter in

the initialization parameter file, or alter it using the MTS_SERVERS parameter of the ALTER SYSTEM statement. Experiment with this limit and monitor shared servers to determine an ideal setting for this parameter.

Setting the MTS Highwater Mark Equal to MTS_MAX_SERVERS This is the first stage in troubleshooting MTS. Performance can degrade if there are not enough shared servers to process all the requests put toward the database.

Check for the initial setting of the maximum number of shared servers. For example:

```
SHOW PARAMETER MTS_MAX_SERVERS
```

Check for the highwater mark for shared servers. For example:

```
SELECT maximum_connections "MAXIMUM_CONNECTIONS",
    servers_started "SERVERS_STARTED", servers_terminated "SERVERS_TERMINATED",
    servers_highwater "SERVERS_HIGHWATER"
FROM V$MTS;
```

The output is:

```
MAXIMUM_CONNECTIONS SERVERS_STARTED SERVERS_TERMINATED SERVERS_HIGHWATER
------------------- --------------- ------------------ -----------------
                 60              30                 30                50
```

Here, HIGHWATER should not be equal to the parameter MTS_MAX_SERVERS.

The other parameters are:

MAXIMUM_CONNECTIONS	The maximum number of connections a single dispatcher can handle.
SERVERS_STARTED	The cumulative number of shared servers that have been started.
SERVERS_TERMINATED	The cumulative number of shared servers that have been terminated.

Increasing the Maximum Number of Shared Servers The shared servers are the processes that perform data access and pass back this information to the dispatchers.

The dispatchers then forward the data to the client process. If there are not enough shared servers to handle all the requests, then the queue backs up (V$QUEUE), and requests take longer to process. However, before you check the V$QUEUE statistics, it is best to first check if you are running out of shared servers.

Find out the amount of free RAM in the system. Examine ps or any other operating system utility to find out the amount of memory a shared server uses. Divide the amount of free RAM by the size of a shared server. This gives you the maximum number of shared servers you can add to your system.

The best way to proceed is to increase the MTS_MAX_SERVERS parameter gradually until you begin to swap. If swapping occurs due to the shared server, then back off the number until swapping stops, or increase the amount of physical RAM.

Because each operating system and application is different, the only way to find out the correct setting for MTS_MAX_SERVERS is through trial and error.

To change the MTS_MAX_SERVERS, first edit the initialization parameter file. Find in the file the parameter MTS_MAX_SERVERS and change it there. Save the file and restart the instance. Remember that setting MTS_SERVERS to MTS_MAX_SERVERS should only be done if you are sure that you will be using the machine at 100% all the time. The general rules are:

- MTS_SERVERS should be set for slightly greater than the expected number of shared servers that will be needed when the database is at an average load.
- MTS_MAX_SERVERS should be set for slightly greater than the expected number of shared servers that will be needed when the database is at an peak load.

See Also: If HIGHWATER does not equal MTS_MAX_SERVERS, then you need to tune MTS. MTS is a complicated configuration and has many points where degradation can occur.

Determining the Optimal Number of Dispatchers and Shared Servers

As mentioned, MTS_SERVERS determines the number of shared servers activated at instance startup. The default setting for MTS_SERVERS is 1 which is the default setting when MTS_DISPATCHERS is specified.

To determine the optimal number of dispatchers and shared servers, consider the number of users typically accessing the database and how much processing each requires. Also consider that user and processing loads vary over time. For example, a customer service system’s load might vary drastically from peak OLTP-oriented daytime use to DSS-oriented nighttime use. System use can also predictably change over longer time periods such as the loads experienced by an accounting system that vary greatly from mid-month to month-end.

If each user makes relatively few requests over a given period of time, then each associated user process is idle for a large percentage of time. In this case, one shared

server process can serve 10 to 20 users. If each user requires a significant amount of processing,then establish a higher ratio of servers to user processes.

In the beginning, it is best to allocate fewer shared servers. Additional shared servers start automatically as needed and are deallocated automatically if they remain idle too long. However, the initial servers always remain allocated, even if they are idle.

If you set the initial number of servers too high, then your system might incur unnecessary overhead. Experiment with the number of initial shared servers and monitor shared servers until you achieve ideal system performance for your typical database activity.

Estimating the Maximum Number of Dispatcher Processes Use values for `MTS_MAX_DISPATCHERS` and `MTS_DISPATCHERS` that are at least equal to the maximum number of concurrent sessions divided by the number of connections per dispatcher. For most systems, a value of 1,000 connections per dispatcher provides good performance.

Disallowing Further MTS Use with Concurrent MTS Use As mentioned, you can use the `SET` option of the `ALTER SYSTEM` statement to alter the number of active, shared servers. To prevent additional users from accessing shared servers, set `MTS_SERVERS` to 0. This temporarily disables additional use of MTS. Re- setting `MTS_SERVERS` to a positive value enables MTS for all current users.

> **See Also:** For information about dispatchers, see the description of the `V$DISPATCHER` and `V$DISPATCHER_RATE` views in the *Oracle8i Reference.* For more information about the `ALTER SYSTEM` statement, see the *Oracle8i SQL Reference.* For more information on changing the number of shared servers, see the *Oracle8i Administrator's Guide.*

Reducing Contention for Parallel Execution Servers

This section describes how to detect and alleviate contention for parallel execution servers when using parallel execution:

- Identifying Contention for Parallel Execution Servers
- Reducing Contention for Parallel Execution Servers

Identifying Contention for Parallel Execution Servers

Statistics in the V$PQ_SYSSTAT view are useful for determining the appropriate number of parallel execution servers for an instance. The statistics that are particularly useful are SERVERS BUSY, SERVERS IDLE, SERVERS STARTED, and SERVERS SHUTDOWN.

Frequently, you cannot increase the maximum number of parallel execution servers for an instance, because the maximum number is heavily dependent upon the capacity of your CPUs and your I/O bandwidth. However, if servers are continuously starting and shutting down, then you should consider increasing the value of the initialization parameter PARALLEL_MIN_SERVERS.

For example, if you have determined that the maximum number of concurrent parallel execution servers that your machine can manage is 100, then you should set PARALLEL_MAX_SERVERS to 100. Next, determine how many parallel execution servers the average parallel operation needs, and how many parallel operations are likely to be executed concurrently. For this example, assume you have two concurrent operations with 20 as the average degree of parallelism. Thus, at any given time there could be 80 parallel execution servers busy on an instance. Thus you should set the PARALLEL_MIN_SERVERS parameter to 80.

Periodically examine V$PQ_SYSSTAT to determine whether the 80 parallel execution servers for the instance are actually busy. To do so, issue the following query:

```
SELECT * FROM V$PQ_SYSSTAT
WHERE STATISTIC = "SERVERS BUSY";
```

The result of this query might look like this:

```
STATISTIC             VALUE
--------------------- -----------
SERVERS BUSY          70
```

Reducing Contention for Parallel Execution Servers

If you find that typically there are fewer than PARALLEL_MIN_SERVERS busy at any given time, then your idle parallel execution servers constitute system overhead that is not being used. Consider decreasing the value of the parameter PARALLEL_MIN_SERVERS. If you find that there are typically more parallel execution servers active than the value of PARALLEL_MIN_SERVERS and the SERVERS STARTED statistic is continuously growing, then consider increasing the value of the parameter PARALLEL_MIN_SERVERS.

Reducing Contention for Redo Log Buffer Latches

Contention for redo log buffer access rarely inhibits database performance. However, Oracle provides methods to monitor and reduce any latch contention that does occur. This section covers:

- Detecting Contention for Redo Log Buffer Latches
- Examining Redo Log Activity
- Reducing Latch Contention

Detecting Contention for Redo Log Buffer Latches

Access to the redo log buffer is regulated by two types of latches: the redo allocation latch and redo copy latches.

The Redo Allocation Latch The redo allocation latch controls the allocation of space for redo entries in the redo log buffer. To allocate space in the buffer, an Oracle user process must obtain the redo allocation latch. Because there is only one redo allocation latch, only one user process can allocate space in the buffer at a time. The single redo allocation latch enforces the sequential nature of the entries in the buffer.

After allocating space for a redo entry, the user process may copy the entry into the buffer. This is called "copying on the redo allocation latch". A process may only copy on the redo allocation latch if the redo entry is smaller than a threshold size.

Redo Copy Latches The user process first obtains the copy latch which allows the process to copy. Then it obtains the allocation latch, performs allocation, and releases the allocation latch. Next the process performs the copy under the copy latch, and releases the copy latch. The allocation latch is thus held for only a very short period of time, as the user process does not try to obtain the copy latch while holding the allocation latch.

If the redo entry is too large to copy on the redo allocation latch, then the user process must obtain a redo copy latch before copying the entry into the buffer. While holding a redo copy latch, the user process copies the redo entry into its allocated space in the buffer and then releases the redo copy latch.

If your computer has multiple CPUs, then your redo log buffer can have multiple redo copy latches. These allow multiple processes to concurrently copy entries to the redo log buffer concurrently.

On single-CPU computers, there should be no redo copy latches, because only one process can be active at once. In this case, all redo entries are copied on the redo allocation latch, regardless of size.

Examining Redo Log Activity

Heavy access to the redo log buffer can result in contention for redo log buffer latches. Latch contention can reduce performance. Oracle collects statistics for the activity of all latches and stores them in the dynamic performance view `V$LATCH`. By default, this table is available only to the user `SYS` and to other users who have `SELECT ANY TABLE` system privilege, such as `SYSTEM`.

Each row in the `V$LATCH` table contains statistics for a different type of latch. The columns of the table reflect activity for different types of latch requests. There is a distinction between the different types of latch requests. The distinction is:

`WILLING-TO-WAIT` — If the latch requested with a willing-to-wait request is not available, then the requesting process waits a short time and requests the latch again. The process continues waiting and requesting until the latch is available.

`IMMEDIATE` — If the latch requested with an immediate request is not available, then the requesting process does not wait, but continues processing.

These columns of the `V$LATCH` view reflect willing-to-wait requests:

`GETS` — Shows the number of successful willing-to-wait requests for a latch.

`MISSES` — Shows the number of times an initial willing-to-wait request was unsuccessful.

`SLEEPS` — Shows the number of times a process waited and requested a latch after an initial willing-to-wait request.

For example, consider the case in which a process makes a willing-to-wait request for a latch that is unavailable. The process waits and requests the latch again and the latch is still unavailable. The process waits and requests the latch a third time and acquires the latch. This activity increments the statistics as follows:

- The `GETS` value increases by one because one request for the latch (the third request) was successful.
- The `MISSES` value increases by one each time because the initial request for the latch resulted in waiting.

- The `SLEEPS` value increases by two because the process waited for the latch twice, once after the initial request and again after the second request.

These columns of the `V$LATCH` table reflect immediate requests:

`IMMEDIATE GETS`	This column shows the number of successful immediate requests for each latch.
`IMMEDIATE MISSES`	This column shows the number of unsuccessful immediate requests for each latch.

Use the following query to monitor the statistics for the redo allocation latch and the redo copy latches over a period of time:

```
SELECT ln.name, gets, misses, immediate_gets, immediate_misses
    FROM v$latch l, v$latchname ln
    WHERE ln.name IN ('redo allocation', 'redo copy')
        AND ln.latch# = l.latch#;
```

The output of this query might look like this:

```
NAME                     GETS       MISSES     IMMEDIATE_GETS  IMMEDIATE_MISSES
------------------------ ---------- ---------- --------------- ----------------
redo allocation          252867     83         0                0
redo copy                0          0          22830            0
```

From the output of the query, calculate the wait ratio for each type of request.

Contention for a latch may affect performance if either of these conditions is true:

- If the ratio of `MISSES` to `GETS` exceeds 1%.
- If the ratio of `IMMEDIATE_MISSES` to the sum of `IMMEDIATE_GETS` and `IMMEDIATE_MISSES` exceeds 1%.

If either of these conditions is true for a latch, then try to reduce contention for that latch. These contention thresholds are appropriate for most operating systems, though some computers with many CPUs may be able to tolerate more contention without performance reduction.

Reducing Latch Contention

Most cases of latch contention occur when two or more Oracle processes concurrently attempt to obtain the same latch. Latch contention rarely occurs on single-CPU computers, where only a single process can be active at once.

Reducing Contention for the Redo Allocation Latch To reduce contention for the redo allocation latch, you should minimize the time that any single process holds the latch. To reduce this time, reduce copying on the redo allocation latch. Decreasing the value of the LOG_SMALL_ENTRY_MAX_SIZE initialization parameter reduces the number and size of redo entries copied on the redo allocation latch.

Reducing Contention for Redo Copy Latches On multiple-CPU computers, multiple redo copy latches allow multiple processes to copy entries to the redo log buffer concurrently. The default value of LOG_SIMULTANEOUS_COPIES is the number of CPUs available to your Oracle instance.

If you observe contention for redo copy latches, then add more latches by increasing the value of LOG_SIMULTANEOUS_COPIES. Consider having twice as many redo copy latches as CPUs available to your Oracle instance.

Reducing Contention for the LRU Latch

The LRU (least recently used) latch controls the replacement of buffers in the buffer cache. For symmetric multiprocessor (SMP) systems, Oracle automatically sets the number of LRU latches to a value equal to one half the number of CPUs on the system. For non-SMP systems, one LRU latch is sufficient.

Contention for the LRU latch can impede performance on SMP machines with a large number of CPUs. You can detect LRU latch contention by querying V$LATCH, V$SESSION_EVENT, and V$SYSTEM_EVENT. To avoid contention, consider bypassing the buffer cache or redesigning the application.

You can specify the number of LRU latches on your system with the initialization parameter DB_BLOCK_LRU_LATCHES. This parameter sets the maximum value for the desired number of LRU latches. Each LRU latch controls a set of buffers; Oracle balances allocation of replacement buffers among the sets.

To select the appropriate value for DB_BLOCK_LRU_LATCHES, consider the following:

- The maximum number of latches is twice the number of CPUs in the system. That is, the value of DB_BLOCK_LRU_LATCHES can range from 1 to twice the number of CPUs.
- A latch should have no less than 50 buffers in its set; for small buffer caches there is no added value if you select a larger number of sets. The size of the buffer cache determines a maximum boundary condition on the number of sets.
- Do not create multiple latches when Oracle runs in *single process* mode. Oracle automatically uses only one LRU latch in single process mode.

- If the workload on the instance is large, then you should have a higher number of latches. For example, if you have 32 CPUs in your system, then choose a number between half the number of CPUs (16) and actual number of CPUs (32) in your system.

> **Note:** You cannot dynamically change the number of sets during the lifetime of the instance.

Reducing Free List Contention

Free list contention can reduce the performance of some applications. This section covers:

- Identifying Free List Contention
- Adding More Free Lists

Identifying Free List Contention

A free list is a list of free data blocks that can be drawn from a number of different extents within the segment. Blocks in free lists contain free space greater than `PCTFREE`. This is the percentage of a block to be reserved for updates to existing rows. In general, blocks included in process free lists for a database object must satisfy the `PCTFREE` and `PCTUSED` constraints.

> **See Also:** For information on free lists, `PCTFREE`, and `PCTUSED`, see *Oracle8i Concepts.*

You can specify the number of process free lists with the `FREELISTS` parameter. The default value of `FREELISTS` is 1. This is the minimum value. The maximum value depends on the data block size. If you specify a value that is too large, an error message informs you of the maximum value. In addition, for each free list, you need to store a certain number of bytes in a block to handle overhead.

> **See Also:** The reserved area and the number of bytes required per free list depend upon your platform. For more information, see your Oracle system-specific documentation.

Contention for free lists is reflected by contention for free data blocks in the buffer cache. You can determine whether contention for free lists is reducing performance by querying the dynamic performance view `V$WAITSTAT`.

Use the following procedure to find the segment names and free lists that have contention:

1. Check V$WAITSTAT for contention on DATA BLOCKS.
2. Check V$SYSTEM_EVENT for BUFFER BUSY WAITS.

   High numbers indicate that some contention exists.
3. In this case, check V$SESSION_WAIT to see, for each buffer busy wait, the values for FILE, BLOCK, and ID.
4. Construct a query as follows to obtain the name of the objects and free lists that have the buffer busy waits:

```
SELECT SEGMENT_NAME, SEGMENT_TYPE
FROM DBA_EXTENTS
WHERE FILE_ID = file
AND BLOCK BETWEEN block_id AND block_id + blocks;
```

   This returns the segment name (*segment)* and type (*type)*.
5. To find the free lists, query as follows:

```
SELECT SEGMENT_NAME, FREELISTS
FROM DBA_SEGMENTS
WHERE SEGMENT_NAME = SEGMENT
AND SEGMENT_TYPE = TYPE;
```

Adding More Free Lists

The ALTER FREELISTS statement lets you modify the FREELIST setting of the existing database objects. To reduce contention for the free lists of a table, use the ALTER FREELISTS statement to add free lists. Set the value of this parameter proportional to the number of processes doing concurrent INSERTs in the steady state.

> **See Also:** For information about using free list groups in a Parallel Server environment, see *Oracle8i Parallel Server Administration, Deployment, and Performance.*

22

Tuning Networks

This chapter introduces networking issues that affect tuning.

This chapter contains the following sections:

- Understanding Connection Models
- Detecting Network Problems
- Solving Network Problems

Understanding Connection Models

The techniques used to determine the source of problems vary depending on the configuration. The three types of configurations are:

- Multi-Threaded Server (MTS) Configuration
- Dedicated Server Configuration
- Pre-Spawned Dedicated Server Configuration

Table 22–1 lists how to tell what type of database configuration you have.

Table 22–1 Database Configurations

Multi-Threaded Server	`LSNRCTL` services lists `dispatchers`.
Dedicated Server	`LSNRCTL` services lists `dedicated servers`.
Pre-Spawn Dedicated Server	`LSNRCTL` services lists `prespawned servers`.

It is possible to connect to dedicated server with a database configured for MTS by placing the parameter (`SERVER = DEDICATED`) in the connect descriptor.

Multi-Threaded Server (MTS) Configuration

Registering the Dispatchers The `LSNRCTL` control utility's `services` statement lists every dispatcher registered with it. This list includes the dispatchers process ID. You can check the alert log to confirm that the dispatcher have been started successfully.

> **Note:** Remember that PMON may take a minute to register the dispatcher with the listener.

For example:

```
lsnrctl services:
LSNRCTL for Solaris: Version 8.1.6.0.0 - Production on 27-MAY-99 13:38:02
(c) Copyright 1999 Oracle Corporation.  All rights reserved.
Connecting to (ADDRESS=(PROTOCOL=TCP)(Host=ecdc2)(Port=1521))
Services Summary...
  ORCL          has 2 service handler(s)
    DEDICATED SERVER established:0 refused:0
      LOCAL SERVER
    DISPATCHER established:0 refused:0 current:0 max:1 state:ready
```

```
    D000 <machine: ecdc2, pid: 16011>
    (ADDRESS=(PROTOCOL=tcp)(DEV=20)(HOST=144.25.216.223)(PORT=55304))

The command completed successfully.
```

Configuring the Initialization Parameter File

- Make sure that the `MTS_DISPATCHER` line is correctly set. For example:

  ```
  MTS_DISPATCHERS =
  "(DESCRIPTION=(ADDRESS=(PROTOCOL=TCP)(HOST=hostname)(PORT=1492)(queuesize=32
  )))
            (DISPATCHERS = 1)
            (LISTENER = alias)
            (SERVICE = servicename)
            (SESSIONS = 1000)
            (CONNECTIONS = 1000)
            (MULTIPLEX = ON)
            (POOL = ON)
            (TICK = 5)"
  ```

  One, and only one, of the following attributes is required: `PROTOCOL`, `ADDRESS`, or `DESCRIPTION`. `ADDRESS` and `DESCRIPTION` provide support for the specification of additional network attributes beyond `PROTOCOL`. In the example above, the entire line with "`DESCRIPTION`" can be substituted by (`PROTOCOL=TCP`). The attributes `DISPATCHERS`, `LISTENER`, `SERVICE`, `SESSIONS`, `CONNECTIONS`, `MULTIPLEX`, `POOL`, and `TICKS` are all optional.

  See Also: For more information on there parameters, see *Oracle8i Reference* and *Net8 Administrator's Guide*.

- Make sure that the optional `MTS_MAX_DISPATCHER` line is correctly set. For example:

  ```
  MTS_MAX_DISPATCHERS = 4
  ```

  This line should reflect the total number of dispatchers you may want to start.

- Make sure that the optional `MTS_MAX_SERVERS` line is correctly set. For example:

  ```
  MTS_MAX_SERVERS = 5
  ```

  This line sets the upper bound on the total number of shared servers PMON can create, based on the peak load of the system. This should be set high enough so

that all requests can be serviced, but not so high that the system swaps if they are reached. The purpose of this parameter is to prevent the server from swapping. Run the following script to see what the highwater mark is for the number of servers running, and then set `MTS_MAX_SERVERS` to more then this.

```
SELECT maximum_connections "MAX CONN", servers_started "STARTED",
    servers_terminated "TERMINATED", servers_highwater "HIGHWATER"
FROM V$MTS;
```

- Make sure that the optional `MTS_SERVERS` line is correctly set. For example:

```
MTS_SERVERS = 5
```

This is the total number of shared servers started when the database is started. It also represents the total number of shared servers PMON tries to keep. It should be the total number of servers expected to always be used when the database is active. `MTS_MAX_SERVERS` is intended to handle peak load.

Registration

Checking the Connections Use the `LSNRCTL` control utility's `services` statement to see if there are excessive connection refusals. Check the listener's log file to see if this is a connection problem. For example:

```
LSNRCTL> set displaymode normal
Service display mode is NORMAL
LSNRCTL> services
Connecting to
(DESCRIPTION=(ADDRESS=(PROTOCOL=ipc)(KEY=net)(QUEUESIZE=32)))
Services Summary...
Service "net.regress.rdbms.dev.us.oracle.com"           has 1 instances.
    Instance "net"
      Status: READY  Total handlers: 3  Relevant handlers: 3
        DEDICATED established:0 refused:0 current:0 max:0 state:ready
            Session: NS
        D001 established:0 refused:0 current:0 max:16383 state:ready

(ADDRESS=(PROTOCOL=tcp)(HOST=dlsun1013.us.oracle.com)(PORT=52217))
            Session: NS
        D000 established:0 refused:0 current:0 max:16383 state:ready

(ADDRESS=(PROTOCOL=tcp)(HOST=dlsun1013.us.oracle.com)(PORT=52216))
            Session: NS
```

Under normal conditions, the number refused should be zero. Shut down the listener, and restart it to erase these statistics. If, after the listener restarts, the refused count is increasing, then the connections are being refused. If the refused count stays at zero, and if the problem you are troubleshooting is occurring, then your problem is not with the connections being refused.

Checking the Connect/Second Rate Connection refusals can occur for many reasons. Examine the listener log to see what the connect per second rate is. Run the listener log analyzer script to check.

The listener is a queue-based process. It receives connect requests from the lower level protocol stack. It has a limited queue stack (which is configurable to the operating system maximum). It can only process one connection at a time, and there is a limit to the number of connections per second the process can handle.

If the rate at which the connect requests arrive exceeds that limit, then the requests will be queued. The queue stack is also limited, but you can configure it. If there are more listener processes, then the requests made against each process will be fewer and, therefore, will be handled more quickly.

Increasing the listener queue is done in the `listener.ora` file. The `listener.ora` file can contain many listeners, each by a different name. It is assumed that only one of those listed is having a problem. If not, then apply this method to all applicable listeners. To increase the listener queue, add (`queuesize =` *`number`*) to the `listener.ora` file. For example:

```
listener =
    (address =
        (protocol = tcp)
        (host = sales-pc)
        (port = 1521)
        (queuesize = 20)
    )
```

> **See Also:** For more information, see *Net8 Administrator's Guide*.

Stop and restart the listener to initialize this new parameter. If you are not currently running an MTS configuration, then you should consider doing so. It is faster for the listener to handle a client request in an MTS configuration than it is in a dedicated server or a pre-spawned dedicated server configuration.

Note: MTS dispatchers also receive connect requests and may also benefit from tuning the queuesize.

The maximum queue size is subject to the maximum size possible for a particular operating system.

Pre-Spawned Dedicated Server Configuration

Pre-spawned (pre-started) processes can improve connect time with a dedicated server. This is particularly true of heavily loaded systems not using multi-threaded servers, where connect time is slow. If this is enabled, then the listener can redirect the connection to an existing process with no wait time whenever a connection request arrives. Connection requests do not have to wait for new processes to be started.

Note: Oracle Corporation recommends that you use a multi-threaded server configuration rather than a pre-spawned dedicated server configuration to solve performance and scalability problems. Use of pre-spawned dedicated servers is recommended only on platforms where MTS is not available.

Checking for the Correct Number of Dedicated Pre-Spawn Servers Determine if the pre-spawn configuration was properly configured and sized for this system.

Pre-spawning dedicated servers have intents. One is to have faster connect times to the database by having the server shadow processes created before the client makes a connect request. Once connected, the listener creates the next shadow process for the next connect. Pre-spawning is also useful in a controlling resource starved system or access into the system. It lets you cap the number of shadow processes that can be pre-spawned. After this limit is reached, all new connections come in as dedicated.

If there is no activity on the database and there are no users connected, then the number of pre-spawn servers is the number listed in the `listener.ora` file for `POOL_SIZE`. Otherwise, depending on the number of connections to the database, they will range from the minimum (`POOL_SIZE`) to the maximum (`PRESPAWN_MAX`). For example:

```
LISTENER =
    (ADDRESS_LIST =(ADDRESS= (PROTOCOL= TCP)(Host= ecdc2)(Port= 1521)))
SID_LIST_LISTENER =
    (SID_LIST =
```

```
        (SID_DESC =
            (ORACLE_HOME = /u01/oracle/product/oracle/8.1.6)
            (SID_NAME = ORCL)
            (PRESPAWN_MAX = 12)
            (PRESPAWN_LIST =
                (PRESPAWN_DESC =
                    (PROTOCOL = TCP)
                    (POOL_SIZE = 1)
                    (TIMEOUT = 1)))))
```

In the above example, with no database activity, there will be one pre-spawn process. During periods of high activity there will be a maximum of 12. After this point, any connect requests that arrive have their shadow processes created in the same manner as a dedicated connection.

To check if there are the correct number of pre-spawn processes, use the `LSNRCTL` utility's `services` statement and the operating system command to list running processes (`ps` on UNIX). For example:

```
lsnrctl services:
LSNRCTL for Solaris: Version 8.1.6.0.0 - Production on 26-MAY-99 18:22:49
(c) Copyright 1999 Oracle Corporation.  All rights reserved.
Connecting to (ADDRESS=(PROTOCOL=TCP)(Host=ecdc2)(Port=1521))
Services Summary...
  ORCL          has 2 service handler(s)
    DEDICATED SERVER established:0 refused:0
      LOCAL SERVER
    PRESPAWNED SERVER established:0 refused:0 current:0 max:1 state:ready
      PID:15587
      (ADDRESS=(PROTOCOL=tcp)(DEV=8)(HOST=144.25.216.223)(PORT=55221))

The command completed successfully

ps -ef | grep oracle
oracle 15587     1  0 17:54:21 ?        0:00 oracleORCL /
        (DESCRIPTION=(COMMAND=prespawn)(PROTOCOL=TCP)(SERVICE_ID=2)(HANDLER_
```

The first statement shows that there is one pre-spawn server process, which is confirmed by the `ps` command.

If there were more pre-spawn servers listed by `ps` than set by the `PRESPAWN_MAX` parameter, then there are processes that are defunct.

If there were other process listed in the `ps` command, like

```
oracle 15634     1  3 18:31:31 ?        0:01 oracleORCL (LOCAL=NO)
```

then there may not be enough pre-spawn servers to handle the load for this configuration. Extra processes like this imply that the maximum number of pre-spawn servers needs to be increased.

There should be, at a minimum, the same number of idle pre-spawn servers as the `POOL` parameter. This can be examined by looking at `LSNRCTL` services to see how many pre-spawn servers have no current connections. For example:

```
lsnrctl services:
LSNRCTL for Solaris: Version 8.1.6.0.0 - Production on 26-MAY-99 18:22:49
(c) Copyright 1999 Oracle Corporation.  All rights reserved.
Connecting to (ADDRESS=(PROTOCOL=TCP)(Host=ecdc2)(Port=1521))
Services Summary...
  ORCL          has 2 service handler(s)
    DEDICATED SERVER established:0 refused:0
      LOCAL SERVER
    PRESPAWNED SERVER established:0 refused:0 current:0 max:1 state:ready
      PID:15587
      (ADDRESS=(PROTOCOL=tcp)(DEV=8)(HOST=144.25.216.223)(PORT=55221))

The command completed successfully
```

Note: The number of current connections to the pre-spawn server is 0 (current:0). This means that it is part of the free pool of pre-spawn. If there are fewer idle pre-spawn than the pool is configured for, then you must be hitting the maximum of pre-spawn.

Determining the Problem To determine if there is a problem with the listener or pre-spawn servers, test to see if the behavior is due to pre-spawn or something else.

Create a listener not configured for pre-spawn. By placing (`SERVER=DEDICATED`) in the connect descriptor, it still connects to a pre-spawn server process. You need to create a listener that does not pre-spawn for this test.

See Also: For more information on setting up `tnsnames.ora` and `listener.ora`, see *Net8 Administrator's Guide*.

Determining if There Enough Physical RAM Determine how much RAM is present so that you do not cause the server to swap when the number of pre-spawn servers is increased.

Find out what the physical size of the pre-spawn server is. On some systems, the command ps gives a false size to a process. Sometimes it gives you the size of the process plus any common memory it shares, like the SGA. Check with the system administrator of the system to get the proper utility.

Find out the amount of RAM that is free in the system. This amount must not include the amount of swap.

Divide the total amount of free RAM by the size of the pre-spawn server process. This gives you an approximate top number of pre-spawn servers that you can add to the system without the fear of it beginning to swap. The actual number of servers to pre-spawn depends on the suspected number of simultaneous connections plus the expected connect rate. The first number determines the setting for PRESPAWN_MAX, while the other number determines the setting for POOLSIZE.

Detecting Network Problems

This section encompasses Local Area Network (LAN) and Wide Area Network (WAN) troubleshooting methods.

Using Dynamic Performance Views

Networks entail overhead that adds a certain amount of delay to processing. To optimize performance, you must ensure that your network throughput is fast, and you should try to reduce the number of messages that must be sent over the network. It can be difficult to measure the delay the network adds.

Three dynamic performance views are useful for measuring the network delay: V$SESSION_EVENT, V$SESSION_WAIT, and V$SESSTAT.

In V$SESSION_EVENT, the AVERAGE_WAIT column indicates the amount of time that Oracle waits between messages. You can use this statistic as a yardstick to evaluate the effectiveness of the network.

In V$SESSION_WAIT, the EVENT column lists the events for which active sessions are waiting. The "sqlnet message from client" wait event indicates that the shared or foreground process is waiting for a message from a client. If this wait event has occurred, then you can check to see whether the message has been sent by the user or received by Oracle.

You can investigate hang-ups by looking at V$SESSION_WAIT to see what the sessions are waiting for. If a client has sent a message, then you can determine whether Oracle is responding to it or is still waiting for it.

In V$SESSTAT you can see the number of bytes that have been received from the client, the number of bytes sent to the client, and the number of calls the client has made.

Understanding Latency and Bandwidth

The most critical aspects of a network that contribute to performance are *latency* and *bandwidth*.

The term latency refers to a time delay; for example, the gap between the time a device requests access to a network and the time it receives permission to transmit.

Bandwidth is the throughput capacity of a network medium or protocol. Variations in the network signals can cause degradation on the network. Sources of degradation can be cables that are too long or wrong cable type. External noise sources, such as elevators, air handlers, or florescent lights, can also cause problems.

Common Network Topologies

Local Area Network Topologies:

- Ethernet
- Fast Ethernet
- 1 Gigabit Ethernet
- Token Ring
- FDDI
- ATM

Wide Area Network Topologies:

- DSL
- ISDN
- Frame Relay
- T-1, T-3, E-1, E-3
- ATM
- SONAT

Table 22–2 lists the most common ratings for various topologies.

Table 22–2 Bandwidth Ratings

Topology or Carrier	Bandwidth
Ethernet	10 Megabits/second
Fast Ethernet	100 Megabits/second
1 Gigabit Ethernet	1 Gigabits/second
Token Ring	16 Megabits/second
FDDI	100 Megabits/second
ATM	155 Megabits/second (OC3), 622 Megabits/second (OC12)
T-1 (US only)	1.544 Megabits/second
T-3 (US only)	44.736 Megabits/second
E-1 (non-US)	2.048 Megabits/second
E-3 (non-US)	34.368 Megabits/second
Frame Relay	Committed Information Rate, which can be up to the carrier speed, but usually is not.
DSL	This can be up to the carrier speed.
ISDN	This can be up to the carrier speed. It is usually used with slower modems.
Dial Up Modems	56 Kilobits/second. It is usually accompanied with data compression for faster throughput.

Solving Network Problems

This section describes several techniques for enhancing performance and solving network problems.

- Finding Bottlenecks
- Dissecting Bottlenecks
- Using Array Interfaces
- Adjusting Session Data Unit Buffer Size
- Using TCP.NODELAY
- Using Connection Manager

> **See Also:** For more information, see *Net8 Administrator's Guide.*

Finding Bottlenecks

The first step in solving network problem is to understand the overall topology. Gather as much information about the network that you can. This kind of information usually manifests itself as a network diagram. Your diagram should contain the types of network technology used in the Local Area Network and the Wide Area Network. It should also contain addresses of the various network segments.

Examine this information. Obvious bottlenecks include:

- Using a dial-up modem (normal modem or ISDN) to access time critical data.
- A frame relay link is running on a T-1, but has a 9.6 Kilobits CIR so that it only reliably transmits up to 9.6 Kilobit's per second and if the rest of the bandwidth is used, then there is a possibly that the data will be lost.
- Data from high speed networks channels through low speed networks.
- There are too many network hops (a router constitutes one hop).
- A 10 Megabit network for a Web site.

There are many problems that can cause a performance breakdown. Follow this checklist:

- Get a Network Sniffer trace.
- Check the following:
  - Is the bandwidth being exceeded on the network, the client, and/or the server?
  - Ethernet collisions.
  - Token ring or FDDI ring beacons.
  - Are there many runt frames?
  - The stability of the WAN links.
- Get a bandwidth utilization chart for frame relay, and see if CIR is being exceeded.
- Is any Quality of Service or packet prioritizing going on?
- Is a firewall in the way somewhere?

If nothing is revealed, then find the network route from the client to the data server. Understanding the travel times on a network gives you an idea as to the time a transaction will take. Client-server communication requires many small packets. High latency on a network slows the transaction down due to the time interval between sending a request and getting the response.

Use trace route (`trcroute` or equivalent) from the client to the server to get address information for each device in the path. For example:

```
tracert usmail05
Tracing route to usmail05.us.oracle.com [144.25.88.200]over a maximum of 30
hops:
  1   <10 ms   <10 ms    10 ms  whq1davis-rtr-749-f1-0-a.us.oracle.com
[144.25.216.1]
  2   <10 ms   <10 ms   <10 ms  whq4op3-rtr-723-f0-0.us.oracle.com
[144.25.252.23]
  3   220 ms   210 ms   231 ms  usmail05.us.oracle.com [144.25.88.200]

Trace complete.
```

Ping each device in turn to get the timings. Use large packets to get the slowest times. Make sure you set the "don't fragment bit" so that routers do not spend time disassembling and reassembling the packet. Also note that the packet size is 1472. This is for Ethernet. Ethernet packets are 1536 octets (actual 8 bit bytes) in size. ICPM packets (this is what ping is designed to use) have 64 octets of header. Evaluate the area where the slowness seems to occur. For example:

```
ping -l 1472 -n 1 -f 144.25.216.1
Pinging 144.25.216.1 with 1472 bytes of data:
Reply from 144.25.216.1: bytes=1472 time<10ms TTL=255

ping -l 1472 -n 1 -f 144.25.252.23
Pinging 144.25.252.23 with 1472 bytes of data:
Reply from 144.25.252.23: bytes=1472 time=10ms TTL=254

ping -l 1472 -n 1 -f 144.25.88.200
Pinging 144.25.88.200 with 1472 bytes of data:
Reply from 144.25.88.200: bytes=1472 time=271ms TTL=253
```

The above example validates trace route. Ideally, you would ping from the workstation to 144.25.216.1, from 144.25.216.1 to 144.25.252.23, then from 144.25.252.23 to 144.25.88.200. This would show the exact latency on each segment traveled.

Dissecting Bottlenecks

This section helps you determine the problem with your bottleneck.

Determine if the Problem is with Net8 or the Network

Net8 tracing reveals whether an error is Oracle-specific or due to conditions that the operating system is passing to the Transparent Network Substrate (Oracle TNS layer).

Enable Net8 tracing at the Oracle server, the listener, and at a client suspected of having the problem you are trying to resolve.

To enable tracing at the server, find the `sqlnet.ora` file for the server and create the following lines in it:

```
TRACE_LEVEL_SERVER = 16
TRACE_UNIQUE_SERVER = ON
```

To enable tracing at the client, find the `sqlnet.ora` file for the client and create the following lines in it:

```
TRACE_LEVEL_CLIENT = 16
TRACE_UNIQUE_CLIENT = ON
```

To enable tracing at the listener, find the `listener.ora` file and create the following line in it:

```
TRACE_LEVEL_listener_name = 16
```

Reproduce the problem, so that you generate traces on the client and server. Now analyze the traces generated.

> **See Also:** For detailed directions on enabling Net8 tracing, see *Net8 Administrator's Guide*. For definitions to Net8 errors noted in the trace file, see *Oracle8i Error Messages*.

If the problem is with the network and not Net8, then you must determine the following:

- Does the problem only occur in one location on the local network?
- Does the problem only occur in one area on the WAN?

For example, perhaps the system is fine in the building where the Data Center is, but it is slow in other buildings that are several miles away.

Not all Oracle error codes represent pure Oracle troubles. `ORA-3113` is the most common error which points to an underlying network problem.

Note: Enabling tracing on the server can generate a large amount of trace, or large number of trace, files. To prevent this, you can set up a separate environment that traces itself. This configuration works for dedicated and pre-spawn connections. First, log into the server's operating system as the Oracle software owner. Create a temporary directory to keep configuration files and trace files that will be created. Copy the `sqlnet.ora`, `listener.ora`, and `tnsnames.ora` to that directory. Edit the `sqlnet.ora` file to enable tracing as above. Add to the `sqlnet.ora` file the following line:

```
TRACE_DIRECTORY_SERVER = temporary directory just created
```

Now, modify the `listener.ora` file and change the listening port (for TCP, other protocols, use a similar technique) to an unused port. You need to make a similar modification to the client's `tnsnames.ora` file for the connect string you will be using for this test.

Set the `TNS_ADMIN` environment to point to the temporary directory. Start the listener. Now all new connections to the new listener send Server traces to this directory. Reproduce the problem.

If you are getting an Oracle error message, then look into the trace file to find the error. For troubleshooting bugs, Net8 trace analysis takes some time to fully find the problem. However, high level simple trace analysis is rather simple.

On Net8, Determine if the Problem is on the Client or the Server

If the problem is with Net8, then use Net8 tracing to show you where the problem lies. If there are errors in the trace files, then do they appear in only the client traces, only in the server traces, or in both?

Errors Only in the Client Trace

The problem is on the client. However, if you are getting `ORA-3113` or `ORA-3114` errors, then the problem is on the server.

Errors Only in the Server Trace or Listener Trace

The problem is on the server. However, if you are getting ORA-3113 or ORA-3114 errors, then the problem is on the client.

Errors in All: Client, Server, and Listener Trace

If you are getting ORA-3113 or ORA-3114 errors, then the problem is on the Network. Troubleshoot the server first. If it is fine, then the client is at fault.

Check if the Server is Configured for MTS

The multi-threaded server (MTS) is an advanced solution for many customers, and it can be more complex to troubleshoot. Check the initialization parameter file for any MTS parameters. Look at the operating system to see if any of the MTS processes are present.

Check for dispatchers by looking for names like ora_d000, ora_d001, etc. For example:

```
ps -ef | grep ora_d
```

Check for shared servers by looking for names like ora_s000, ora_s001, etc. For example:

```
ps -ef | grep ora_s
```

> **See Also:** For more information on tuning the multi-threaded server, see "Multi-Threaded Server (MTS) Configuration" on page 22-2. For more information on MTS concepts and parameters, see *Oracle8i Concepts* and *Net8 Administrator's Guide*.

Using Array Interfaces

Reduce network calls by using array interfaces. Instead of fetching one row at a time, it is more efficient to fetch ten rows with a single network round trip.

> **See Also:** For more information on array interfaces, see *Oracle Call Interface Programmer's Guide*.

Adjusting Session Data Unit Buffer Size

Before sending data across the network, Net8 buffers data into the Session Data Unit (SDU). It sends the data stored in this buffer when the buffer is full or when an application tries to read the data. When large amounts of data are being retrieved and when packet size is consistently the same, it may speed retrieval to adjust the default SDU size.

Optimal SDU size depends on the normal transport size. Use a sniffer to find out the frame size, or set tracing on to its highest level to check the number of packets sent and received and to determine whether they are fragmented. Tune your system to limit the amount of fragmentation.

Use Net8 Assistant to configure a change to the default SDU size on both the client and the server; SDU size should generally be the same on both.

> **See Also:** For more information, see *Net8 Administrator's Guide*.

Using TCP.NODELAY

When a session is established, Net8 packages and sends data between server and client using packets. Use the `TCP.NODELAY` parameter in the `protocol.ora` file, which causes packets to be flushed on to the network more frequently. If you are streaming large amounts of data, then there is no buffering and hence no delay.

Although Net8 supports many networking protocols, TCP tends to have the best scalability.

> **See Also:** For more information on `TCP.NODELAY`, see your platform-specific Oracle documentation.

Using Connection Manager

In Net8, you can use the Connection Manager to conserve system resources by multiplexing. *Multiplexing* means funneling many client sessions through a single transport connection to a server destination. In this way, you can increase the number of sessions that a process can handle. This applies only to MTS configurations.

Alternately, you can use Connection Manager to control client access to dedicated servers. In addition, Connection Manager provides multiple protocol support allowing a client and server with different networking protocols to communicate.

> **See Also:** For more information on Connection Manager, see *Net8 Administrator's Guide*.

23

Tuning the Operating System

This chapter explains how to tune the operating system for optimal performance of the Oracle server.

This chapter contains the following sections:

- Understanding Operating System Performance Issues
- Detecting Operating System Problems
- Solving Operating System Problems

See Also: In addition to information in this chapter, see your operating system specific documentation.

Understanding Operating System Performance Issues

Operating system performance issues commonly involve process management, memory management, and scheduling. If you tuned the Oracle instance and you still need better performance, then verify your work or try to reduce system time. Make sure that there is enough I/O bandwidth, CPU power, and swap space. Do not expect, however, that further tuning of the operating system will have a significant effect on application performance. Changes in the Oracle configuration or in the application are likely to make a more significant difference in operating system efficiency than simply tuning the operating system.

For example, if your application experiences excessive buffer busy waits, then the number of system calls will increase. If you reduce the buffer busy waits by tuning the application, then the number of system calls will decrease. Similarly, if you turn on the Oracle initialization parameter `TIMED_STATISTICS`, then the number of system calls will increase. If you turn it off, then system calls will decrease.

> **See Also:** For detailed information, see your Oracle platform-specific documentation and your operating system vendor's documentation.

Operating System and Hardware Caches

Operating systems and device controllers provide data caches that do not directly conflict with Oracle's own cache management. Nonetheless, these structures can consume resources while offering little or no benefit to performance. This is most noticeable on a UNIX system that has the database files in the UNIX file store: by default all database I/O goes through the file system cache. On some UNIX systems, direct I/O is available to the filestore. This arrangement allows the database files to be accessed within the UNIX file system, bypassing the file system cache. It saves CPU resources and allows the file system cache to be dedicated to non-database activity, such as program texts and spool files.

This problem does not occur on NT. All file requests by the database bypass the caches in the file system.

Raw Devices

Evaluate the use of raw devices on your system. Using raw devices may involve a significant amount of work, but may also provide significant performance benefits.

Raw devices impose a penalty on full table scans, but may be essential on UNIX systems if the implementation does not support "write through" cache. The UNIX file system accelerates full table scans by reading ahead when the server starts

requesting contiguous data blocks. It also caches full table scans. If your UNIX system does not support the write through option on writes to the file system, then it is essential that you use raw devices to ensure that at commit and checkpoint, the data that the server assumes is safely established on disk is actually there. If this is not the case, then recovery from a UNIX operating system crash may not be possible.

Raw devices on NT are similar to UNIX raw devices; however, all NT devices support write through cache.

> **See Also:** For a discussion on raw devices versus UNIX file system (UFS), see Chapter 20, "Tuning I/O".

Process Schedulers

Many processes, or "threads" on NT systems, are involved in the operation of Oracle. They all access the shared memory resources in the SGA.

Be sure that all Oracle processes, both background and user processes, have the same process priority. When you install Oracle, all background processes are given the default priority for your operating system. Do not change the priorities of background processes. Verify that all user processes have the default operating system priority.

Assigning different priorities to Oracle processes may exacerbate the effects of contention. Your operating system may not grant processing time to a low-priority process if a high-priority process also requests processing time. If a high-priority process needs access to a memory resource held by a low-priority process, then the high-priority process may wait indefinitely for the low-priority process to obtain the CPU, process the request, and release the resource.

Additionally, do not bind Oracle background processes to CPUs. This may cause the bound processes to be CPU-starved. This is especially the case when binding processes that fork off operating system threads. In this case, the parent process and all its threads will bind to the CPU.

Operating System Resource Managers

Some platforms provide operating system resource managers. These are designed to reduce the impact of peak load use patterns by prioritizing access to system resources. They usually implement administrative policies that govern which resources users can access, and how much of those resources each user is permitted to consume.

Operating system resource managers are different from domains or other similar facilities. Domains provide one or more completely separated environments within one system. Disk, CPU, memory, and all other resources are dedicated to each domain, and cannot be accessed from any other domain. Other similar facilities completely separate just a portion of system resources into different areas, usually separate CPU and/or memory areas. Like domains, the separate resource areas are dedicated only to the processing assigned to that area; processes cannot migrate across boundaries. Unlike domains, all other resources (usually disk) are accessed by all partitions on a system.

Oracle runs within domains, as well as within these other less complete partitioning constructs, provided that the allocation of partitioned memory (RAM) resources is fixed, not dynamic. Deallocating RAM to enable a memory board replacement is an example of a dynamically changing memory resource; therefore, this is an example of an environment in which Oracle is not supported.

Note: Oracle is not supported in any resource partitioned environment in which memory resources are assigned dynamically.

Operating system resource managers prioritize resource allocation within a global pool of resources, usually a domain or an entire system. Processes are assigned to groups, which are in turn assigned resources anywhere within the resource pool.

Warning: When running under operating system resource managers, Oracle is supported only when each instance is assigned to a dedicated operating system resource manager group or managed entity. Also, the dedicated entity running all the instance's processes must run at one priority (or resource consumption) level. Management of individual Oracle processes at different priority levels is *not* supported. Severe consequences, including instance crashes, can result.

Warning: Oracle is not supported for use with any operating system resource manager's memory management and allocation facility.

Warning: Oracle Database Resource Manager, which provides resource allocation capabilities within an Oracle instance, cannot be used with any operating system resource manager.

See Also: For a complete list of operating system resource management and resource allocation/deallocation features that work with Oracle and Oracle Database Resource Manager, see your systems vendor and your Oracle representative. Note that Oracle does not certify these system features for compatibility with specific release levels.

For more information about Oracle Database Resource Manager, see *Oracle8i Concepts* and *Oracle8i Administrator's Guide*.

Detecting Operating System Problems

The key statistics to extract from any operating system monitor are:

- CPU load
- Device queues
- Network activity (queues)
- Memory management (paging/swapping)

Examine CPU use to determine the ratio between the time spent running in application mode and the time spent running in operating system mode. Look at run queues to see how many processes are runable and how many system calls are being executed. See if paging or swapping is occuring, and check the number of I/Os being performed and the scan rate.

See Also: For more information, see your Oracle platform-specific documentation and your operating system vendor's documentation.

Solving Operating System Problems

This section provides hints for tuning various systems by explaining the following topics:

- Performance on UNIX-Based Systems
- Performance on NT Systems
- Performance on Mainframe Computers

Familiarize yourself with platform-specific issues so you know what performance options your operating system provides. For example, some platforms have post

wait drivers that allow you to map system time and thus reduce system calls, enabling faster I/O.

> **See Also:** For more information, see your Oracle platform-specific documentation and your operating system vendor's documentation.

Performance on UNIX-Based Systems

On UNIX systems, try to establish a good ratio between the amount of time the operating system spends fulfilling system calls and doing process scheduling, and the amount of time the application runs. Your goal should be running 60% to 75% of the time in application mode, and 25% to 40% of the time in operating system mode. If you find that the system is spending 50% of its time in each mode, then determine what is wrong.

The ratio of time spent in each mode is only a symptom of the underlying problem, which might involve:

- Swapping
- Executing too many O/S system calls
- Running too many processes

If such conditions exist, then there is less time available for the application to run. The more time you can release from the operating system side, the more transactions your application can perform.

Performance on NT Systems

On NT systems, as with UNIX-based systems, you should establish an appropriate ratio between time in application mode and time in system mode. On NT you can easily monitor many factors with Performance Monitor: CPU, network, I/O, and memory are all displayed on the same graph, to assist you in avoiding bottlenecks in any of these areas.

Performance on Mainframe Computers

Consider the paging parameters on a mainframe, and remember that Oracle can exploit a very large working set of parameters.

Free memory in VAX/VMS environments is actually memory that is not mapped to any operating system process. On a busy system, free memory likely contains a page belonging to one or more currently active process. When that access occurs, a

"soft page fault" takes place, and the page is included in the working set for the process. If the process cannot expand its working set, then one of the pages currently mapped by the process must be moved to the free set.

Any number of processes may have pages of shared memory within their working sets. The sum of the sizes of the working sets can thus markedly exceed the available memory. When the Oracle server is running, the SGA, the Oracle kernel code, and the Oracle Forms runtime executable are normally all sharable and account for perhaps 80% or 90% of the pages accessed.

Adding more buffers is not necessarily better. Each application has a threshold number of buffers at which the cache hit ratio stops rising. This is typically quite low (approximately 1500 buffers). Setting higher values simply increases the management load for both Oracle and the operating system.

24

Tuning Instance Recovery Performance

This chapter offers guidelines for tuning instance recovery.

This chapter contains the following sections:

- Understanding Instance Recovery
- Tuning the Duration of Instance and Crash Recovery
- Monitoring Instance Recovery
- Tuning the Phases of Instance Recovery

Understanding Instance Recovery

Instance and crash recovery are the automatic application of redo log records to Oracle data blocks after a crash or system failure. If a single instance database crashes, or if all instances of an Oracle Parallel Server configuration crash, then Oracle performs instance recovery at the next startup. If one or more instances of an Oracle Parallel Server configuration crash, then a surviving instance performs recovery.

Instance and crash recovery occur in two phases. In phase one, Oracle applies all committed and uncommitted changes in the redo log files to the affected datablocks. In phase two, Oracle applies information in the rollback segments to undo changes made by uncommitted transactions to the data blocks.

How Oracle Applies Redo Log Information

During normal operations, Oracle's DBW*n* processes periodically write dirty buffers, or buffers that have in-memory changes, to disk. Periodically, Oracle records the highest system change number (SCN) of all changes to blocks, such that all data blocks with changes below that SCN have been written to disk by DBW*n*. This SCN is the *checkpoint*.

Records that Oracle appends to the redo log file after the change record that the checkpoint refers to are changes that Oracle has not yet written to disk. If a failure occurs, then only redo log records containing changes at SCNs higher than the checkpoint need to be replayed during recovery.

The duration of recovery processing is directly influenced by the number of data blocks that have changes at SCNs higher than the SCN of the checkpoint. For example, Oracle recovers a redo log with 100 entries affecting one data block faster than it recovers a redo log with 10 entries for 10 different data blocks. This is because for each log record processed during recovery, the corresponding data block (if it is not already in memory) must be read from disk by Oracle, so that the change represented by the redo log entry can be applied to that block.

Trade-offs of Minimizing Recovery Duration

The principal means of balancing the duration of instance recovery and daily performance is by influencing how aggressively Oracle advances the checkpoint. If you force Oracle to keep the checkpoint only a few blocks behind the most recent redo log record, then you minimize the number of blocks Oracle processes during recovery.

The trade-off for having minimal recovery time, however, is increased performance overhead for normal database operations. If daily operational efficiency is more important than minimizing recovery time, then decreasing the frequency of writes to the datafiles increases instance recovery time.

Tuning the Duration of Instance and Crash Recovery

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

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

The Oracle8*i* Enterprise Edition also offers fast-start fault recovery functionality to control instance recovery. This reduces the roll forward time by making it bounded and predictable, and it also eliminates the time required perform rollback. The foundation of fast-start fault recovery is fast-start checkpointing architecture. Instead of the conventional periodic checkpointing, as performed in earlier versions of Oracle, fast-start checkpointing occurs continuously, advancing the checkpoint time as blocks are written. Fast-start checkpointing always writes the oldest modified block first, ensuring that every write allows the checkpoint time to be advanced. Administrators specify a target (bounded) time to complete the roll forward phase of recovery, and Oracle automatically varies the checkpoint writes to meet that target.

Using Initialization Parameters to Influence Recovery Time

During recovery, Oracle performs two main tasks:

- Reads redo logs to determine what has been changed.
- Reads data blocks to determine whether to apply changes.

Fast-start checkpointing eliminates bulk writes and the resultant I/O spikes that occur with conventional checkpointing, yielding smooth and fast ongoing performance. Continuous advancement reduces roll forward by half, compared to conventional checkpoints at the same transaction rate. Administrators can specify a

bound on the time to do roll forward, rather than specifying the frequency of checkpoints.

You can use three initialization parameters to influence how aggressively Oracle advances the checkpoint, as shown in Table 24–1:

Table 24–1 Initialization Parameters Influencing Checkpoints

Parameter	Purpose
LOG_CHECKPOINT_TIMEOUT	Limits the number of seconds between the most recent redo record and the checkpoint.
LOG_CHECKPOINT_INTERVAL	Limits the number of redo records between the most recent redo record and the checkpoint.
FAST_START_IO_TARGET	Limits instance recovery time by controlling the number of data blocks Oracle processes during instance recovery.

Note: The FAST_START_IO_TARGET parameter is only available with the Oracle8*i* Enterprise Edition.

Using LOG_CHECKPOINT_TIMEOUT

Set the initialization parameter LOG_CHECKPOINT_TIMEOUT to a value *n* (where *n* is an integer) to require that the latest checkpoint position follow the most recent redo block by no more than *n* seconds. In other words, at most, *n* seconds worth of logging activity can occur between the most recent checkpoint position and the end of the redo log. This forces the checkpoint position to keep pace with the most recent redo block

You can also interpret LOG_CHECKPOINT_TIMEOUT as specifying an upper bound on the time a buffer can be dirty in the cache before DBW*n* must write it to disk. For example, if you set LOG_CHECKPOINT_TIMEOUT to 60, then no buffers remain dirty in the cache for more than 60 seconds. The default value for LOG_CHECKPOINT_TIMEOUT is 1800.

Note: The minimum value for LOG_CHECKPOINT_TIMEOUT in the Standard Edition is 900, or 15 minutes. If you set the value below 900 in the Standard Edition, then Oracle rounds it to 900.

Using LOG_CHECKPOINT_INTERVAL

Set the initialization parameter LOG_CHECKPOINT_INTERVAL to a value *n* (where *n* is an integer) to require that the checkpoint position never follow the most recent redo block by more than *n* blocks. In other words, at most *n* redo blocks can exist between the checkpoint position and the last block written to the redo log. In effect, you are limiting the amount of redo blocks that can exist between the checkpoint and the end of the log.

Oracle limits the maximum value of LOG_CHECKPOINT_INTERVAL to 90% of the smallest log to ensure that the checkpoint advances into the current log before that log fills and a log switch is attempted.

LOG_CHECKPOINT_INTERVAL is specified in redo blocks. Redo blocks are the same size as operating system blocks. Use the LOG_FILE_SIZE_REDO_BLKS column in V$INSTANCE_RECOVERY to see the number of redo blocks corresponding to 90% of the size of the smallest log file.

Using FAST_START_IO_TARGET

Note: The initialization parameter FAST_START_IO_TARGET and fast-start checkpointing are only available with the Oracle8*i* Enterprise Edition.

Oracle recommends using fast-start checkpointing to control the duration of the roll forward phase of recovery. This behavior is controlled by the FAST_START_IO_TARGET parameter. The parameter, DB_BLOCK_MAX_DIRTY_TARGET, is an Oracle8 parameter used to provide more limited control over roll forward duration, and it is included in Oracle8*i* only for backward compatibility.

Set this parameter to *n*, where *n* is an integer limiting to *n* the number of buffers that Oracle processes during crash or instance recovery. Because the number of I/Os to be processed during recovery correlates closely to the duration of recovery, the FAST_START_IO_TARGET parameter gives you the most precise control over the duration of recovery.

FAST_START_IO_TARGET advances the checkpoint, because DBW*n* uses the value of FAST_START_IO_TARGET to determine how much writing to do. Assuming that users are making many updates to the database, a low value for this parameter forces DBW*n* to write changed buffers to disk. As the changed buffers are written to disk, the checkpoint advances.

The smaller the value of FAST_START_IO_TARGET, the better the recovery performance, because fewer blocks require recovery. If you use smaller values for this parameter, however, then you impose higher overhead during normal processing, because DBW*n* must write more buffers to disk more frequently.

> **See Also:** For more information, see "Estimating Recovery Time" on page 24-4 and "Calculating Performance Overhead" on page 24-11. For more information on tuning checkpoints, see Chapter 20, "Tuning I/O". For more information about initialization parameters, see the *Oracle8i Reference*.

Using Redo Log Size to Influence Checkpointing Frequency

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

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

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

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

> **See Also:** For information on using the V$INSTANCE_RECOVERY view to tune instance recovery, see "Estimating Recovery Time" on page 24-10.

Using SQL Statements to Initiate Checkpoints

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

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

> **See Also:** For more information about these statements, see the *Oracle8i SQL Reference*.

Monitoring Instance Recovery

Use the V$INSTANCE_RECOVERY view to see your current recovery parameter settings. You can also use statistics from this view to calculate which parameter has the greatest influence on checkpointing. V$INSTANCE_RECOVERY contains the columns shown in Table 24–2.

Table 24–2 V$INSTANCE_RECOVERY View

Column	Description
RECOVERY_ESTIMATED_IOS	The estimated number of blocks that would be processed during recovery. This estimate is based upon FAST_START_IO_TARGET, and it is not valid unless FAST_START_IO_TARGET is driving checkpointing behavior.
ACTUAL_REDO_BLKS	Current number of redo blocks required for recovery.
TARGET_REDO_BLKS	Goal for the maximum number of redo blocks to be processed during recovery. This value is the minimum of the next 4 columns.
LOG_FILE_SIZE_REDO_BLKS	Number of redo blocks to be processed during recovery to guarantee that a log switch never has to wait for a checkpoint. This is 90% of the smallest log file.
LOG_CHKPT_TIMEOUT_REDO_BLKS	Number of redo blocks that must be processed during recovery to satisfy LOG_CHECKPOINT_TIMEOUT.

Table 24–2 V$INSTANCE_RECOVERY View

Column	Description
LOG_CHKPT_INTERVAL_REDO_BLKS	Number of redo blocks that must be processed during recovery to satisfy LOG_CHECKPOINT_INTERVAL.
FAST_START_IO_TARGET_REDO_BLKS	Number of redo blocks that must be processed during recovery to satisfy FAST_START_IO_TARGET.

The value appearing in the TARGET_REDO_BLKS column equals a value appearing in another column in the view. This other column corresponds to the parameter or log file that is determining the maximum number of redo blocks that Oracle processes during recovery. The setting for the parameter in this column is imposing the heaviest requirement on redo block processing.

> **See Also:** For more information on the V$INSTANCE_RECOVERY view, see the *Oracle8i Reference*.

Determining the Strongest Checkpoint Influence

For example, assume your initialization parameter settings are as follows:

```
FAST_START_IO_TARGET = 1000
LOG_CHECKPOINT_TIMEOUT = 1800 # default
LOG_CHECKPOINT_INTERVAL = 0# default: disabled interval checkpointing
```

You execute the following query:

```
SELECT * FROM V$INSTANCE_RECOVERY;
```

Oracle responds with the following:

RECOVERY_ ESTIMATED_ IOS	ACTUAL_REDO_ BLKS	TARGET_REDO_ BLKS	LOG_FILE_ SIZE_REDO_ BLKS	LOG_CHKPT_ TIMEOUT_ REDO_BLKS	LOG_CHKPT_ INTERVAL_ REDO_BLKS	FAST_START_IO_ TARGET_REDO_ BLKS
1025	6169	4215	55296	35485	4294967295	4215

```
1 row selected.
```

As you can see by the values in the last three columns, the FAST_START_IO_TARGET parameter places heavier recovery demands on Oracle than the other two parameters. It requires that Oracle process no more than 4215 redo blocks during recovery. The LOG_FILE_SIZE_REDO_BLKS column indicates that Oracle can process up to 55,296 blocks during recovery, so the log file size is not the heaviest influence on checkpointing.

Note: The value for LOG_CHKPT_INTERVAL_REDO_BLKS, 4294967295, corresponds to the maximum possible value indicating that this column does not have the greatest influence over checkpointing.

The TARGET_REDO_BLKS column shows the smallest value of the last five columns. This shows the parameter or condition that exerts the heaviest requirement for Oracle checkpointing. In this example, the FAST_START_IO_TARGET parameter is the strongest influence with a value of 4215.

Assume you make several updates to the database and query V$INSTANCE_RECOVERY three hours later. Oracle responds with the following:

RECOVERY_ ESTIMATED_ IOS	ACTUAL_ REDO_BLKS	TARGET_ REDO_BLKS	LOG_FILE_ SIZE_REDO_ BLKS	LOG_CHKPT_ TIMEOUT_ REDO_BLKS	LOG_CHKPT_ INTERVAL_ REDO_BLKS	FAST_START_ IO_TARGET_ REDO_BLKS
1022	916	742	55296	44845	4294967295	742

1 row selected.

FAST_START_IO_TARGET is still exerting the strongest influence over checkpointing behavior, although the number of redo blocks corresponding to this target has changed dramatically. This change is not due to a change in FAST_START_IO_TARGET or the corresponding RECOVERY_ESTIMATED_IOS. Instead, this indicates that operations requiring I/O in the event of recovery are more frequent in the redo log, so fewer redo blocks now correspond to the same FAST_START_IO_TARGET.

Assume you decide that FAST_START_IO_TARGET is placing an excessive limit on the maximum number of redo blocks that Oracle processes during recovery. You adjust FAST_START_IO_TARGET to 8000, set LOG_CHECKPOINT_TIMEOUT to 60, and perform several updates. You reissue the query to V$INSTANCE_RECOVERY and Oracle responds with:

RECOVERY_ ESTIMATED_ IOS	ACTUAL_ REDO_BLKS	TARGET_ REDO_BLKS	LOG_FILE_ SIZE_REDO_ BLKS	LOG_CHKPT_ TIMEOUT_ REDO_BLKS	LOG_CHKPT_ INTERVAL_ REDO_BLKS	FAST_START_ IO_TARGET_ REDO_BLKS
1640	6972	6707	55296	6707	4294967295	10338

1 row selected.

Because the TARGET_REDO_BLKS column value of 6707 corresponds to the value in the LOG_CHKPT_TIMEOUT_REDO_BLKS column, LOG_CHECKPOINT_TIMEOUT is now exerting the most influence over checkpointing behavior.

Estimating Recovery Time

Use statistics from the V$INSTANCE_RECOVERY view to estimate recovery time using the following formula:

$$\frac{\textit{RECOVERY_ESTIMATED_JOBS}}{\textit{Maximum I/Os per second that your system can perform}}$$

For example, if RECOVERY_ESTIMATED_IOS is 2500, and the maximum number of writes your system performs is 500 per second, then recovery time is 5 seconds. Note the following restrictions:

- The value for the maximum I/Os per second that the system can perform is difficult to measure accurately. You can estimate this value by measuring the total number of reads and writes your system can perform under a peak load. The V$FILESTAT view provides information on the number of physical reads and writes performed since the instance started. Measure these values over a set time interval, and then divide this by the time interval to estimate your system's maximum I/Os per second. The following query can be used to measure the total I/Os since the instance started:

  ```
  SELECT sum(PHYBLKRD+PHYBLKWRT)
  FROM v$filestat;
  ```

- There is no guarantee the system will sustain the I/O rate during recovery.
- This estimate for recovery time is only valid when FAST_START_IO_TARGET is both enabled and when this parameter is the determining influence on checkpointing behavior.

To adjust recovery time, change the initialization parameter that has the most influence over checkpointing. Use the V$INSTANCE_RECOVERY view as described in "Monitoring Instance Recovery" on page 24-7 to determine which parameter to adjust. Then, either adjust the parameter to decrease or increase recovery time as required.

Adjusting Recovery Time: Example Scenario

For example, assume as in "Determining the Strongest Checkpoint Influence" on page 24-8 that your initialization parameter settings are the following:

```
FAST_START_IO_TARGET = 1000
LOG_CHECKPOINT_TIMEOUT = 1800 # default
LOG_CHECKPOINT_INTERVAL = 0 # default: disabled interval checkpointing
```

You execute the following query:

```
SELECT * FROM V$INSTANCE_RECOVERY;
```

Oracle responds with the following:

RECOVERY_ ESTIMATED_ IOS	ACTUAL_ REDO_BLKS	TARGET_ REDO_BLKS	LOG_FILE_ SIZE_REDO_ BLKS	LOG_CHKPT_ TIMEOUT_ REDO_BLKS	LOG_CHKPT_ INTERVAL_ REDO_BLKS	FAST_START_ IO_TARGET_ REDO_BLKS
1025	6169	4215	55296	35485	4294967295	4215

1 row selected.

You calculate recovery time using the formula on page 24-10, where RECOVERY_ ESTIMATED_IOS is 1025 and the maximum I/Os per second the system can perform is 500:

$$\frac{1025}{500} = 2.05$$

You decide you can afford slightly more than 2.05 seconds of recovery time: constant access to the data is not critical. You increase the value for the parameter FAST_START_IO_TARGET to 2000 and perform several updates. You then reissue the query and Oracle displays:

RECOVERY_ ESTIMATED_ IOS	ACTUAL_ REDO_BLKS	TARGET_ REDO_BLKS	LOG_FILE_ SIZE_REDO_ BLKS	LOG_CHKPT_ TIMEOUT_ REDO_BLKS	LOG_CHKPT_ INTERVAL_ REDO_BLKS	FAST_START_ IO_TARGET_ REDO_BLKS
2007	8301	8012	55296	40117	4294967295	8012

1 row selected.

Recalculate recovery time using the same formula:

$$\frac{2007}{500} = 4.01$$

You have increased your recovery time by 1.96 seconds. If you can afford more time, then repeat the procedure until you arrive at an acceptable recovery time.

Calculating Performance Overhead

To calculate performance overhead, use the V$SYSSTAT view. For example, assume you execute the following query:

```
SELECT NAME, VALUE FROM V$SYSSTAT
WHERE NAME IN ( 'PHYSICAL READS', 'PHYSICAL WRITES',);
```

Oracle responds with the following:

```
NAME                                    VALUE
physical reads                           2376
physical writes                         14932
physical writes non checkpoint          11165
3 rows selected.
```

The first row shows the number of data blocks retrieved from disk. The second row shows the number of data blocks written to disk. The last row shows the value of the number of writes to disk that would occur if you turned off checkpointing.

Use this data to calculate the overhead imposed by setting the `FAST_START_IO_TARGET` initialization parameter. To effectively measure the percentage of extra writes, mark the values for these statistics at different times, `t_1` and `t_2`. Use the following formula where the variables stand for the following:

Variable	Definition
`*_1`	Value of prefixed variable at time `t_1`, which is any time after the database has been running for a while
`*_2`	Value of prefixed variable at time `t_2`, which is later than `t_1` and not immediately after changing any of the checkpoint parameters
`PWNC`	Physical writes non checkpoint
`PW`	Physical writes
`PR`	Physical reads
`EIO`	Percentage of estimated extra I/Os generated by enabling checkpointing

Calculate the percentage of extra I/Os generated by fast-start checkpointing using this formula:

$$[((PW_2 - PW_1) - (PWNC_2 - PWNC_1)) / ((PR_2 - PR_1) + (PW_2 - PW_1))] \times 100\% = EIO$$

It can take some time for database statistics to stabilize after instance startup or dynamic initialization parameter modification. After such events, wait until all blocks age out of the buffer cache at least once before taking measurements.

If the percentage of extra I/Os is too high, then increase the value for FAST_START_IO_TARGET. Adjust this parameter until you get an acceptable value for the RECOVERY_ESTIMATED_IOS in V$INSTANCE_RECOVERY as described in "Determining the Strongest Checkpoint Influence" on page 24-8.

The number of extra writes caused by setting FAST_START_IO_TARGET to a non-zero value is application-dependent. An application that repeatedly modifies the same buffers incurs a higher write penalty because of fast-start checkpointing than an application that does not. The extra write penalty is not dependent on cache size.

Calculating Performance Overhead: Example Scenario

As an example, assume your initialization parameter settings are:

```
FAST_START_IO_TARGET = 2000
LOG_CHECKPOINT_TIMEOUT = 1800 # default
LOG_CHECKPOINT_INTERVAL = 0 # default: disabled interval checkpointing
```

After the statistics stabilize, you issue this query on V$SYSSTAT:

```
SELECT NAME, VALUE FROM V$SYSSTAT
WHERE NAME IN ('PHYSICAL READS', 'PHYSICAL WRITES',
'PHYSICAL WRITES NON CHECKPOINT');
```

Oracle responds with:

```
Name                                 Value
physical reads                        2376
physical writes                      14932
physical writes non checkpoint       11165
3 rows selected.
```

After making updates for a few hours, you re-issue the query and Oracle responds with:

```
Name                                 Value
physical reads                        3011
physical writes                      17467
physical writes non checkpoint       13231
3 rows selected.
```

Substitute the values from your select statements in the formula as described on page 24-11 to determine how much performance overhead you are incurring:

[((17467 - 14932) - (13231 - 11165)) / ((3011 - 2376) + (17467 - 14932))] x 100% = 14.8%

As the result indicates, enabling fast-start checkpointing generates about 15% more I/O than would be required had you not enabled fast-start checkpointing. After calculating the extra I/O, you decide you can afford more system overhead if you decrease recovery time.

To decrease recovery time, reduce the value for the parameter `FAST_START_IO_TARGET` to 1000. After items in the buffer cache age out, calculate `V$SYSSTAT` statistics across a second interval to determine the new performance overhead. Query `V$SYSSTAT`:

```
SELECT NAME, VALUE FROM V$SYSSTAT
WHERE NAME IN ('PHYSICAL READS', 'PHYSICAL WRITES',
'PHYSICAL WRITES NON CHECKPOINT');
```

Oracle responds with:

```
Name                                 Value
physical reads                        4652
physical writes                      28864
physical writes non checkpoint       21784
3 rows selected.
```

After making updates, re-issue the query and Oracle responds with:

```
Name                                 Value
physical reads                        6000
physical writes                      35394
physical writes non checkpoint       26438
3 rows selected.
```

Calculate how much performance overhead you are incurring using the values from your two `SELECT` statements:

[(35394 - 28864) - (26438 - 21784)) / ((6000 - 4652) + (35394 - 28864))] x 100% = 23.8%

After changing the parameter, the percentage of I/Os performed by Oracle is now about 24% more than it would be if you disabled fast-start checkpointing.

Tuning the Phases of Instance Recovery

The work required to do roll forward processing is proportional to the rate of change to the database (update transactions per second) and the time between which consistent snapshots, or checkpoints, of the database are made. The work required to do roll back is proportional to the number and size of uncommitted

transactions when the system fault occurred. The total recovery time is the sum of time to do roll forward and the time to do roll back.

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

This section contains the following topics:

- Tuning the Rolling Forward Phase
- Tuning the Rolling Back Phase

Tuning the Rolling Forward Phase

Use parallel block recovery to tune the roll forward phase of recovery. Parallel block recovery uses a division of labor approach to allocate different processes to different data blocks during the roll forward phase of recovery. For example, during recovery the redo log is read, and blocks that require redo application are parsed out. These blocks are subsequently distributed evenly to all recovery slaves to be read into the buffer cache. Crash, instance, and media recovery of many datafiles on different disk drives are good candidates for parallel block recovery.

Use the `RECOVERY_PARALLELISM` initialization parameter to specify the number of concurrent recovery processes for instance or media recovery operations. Because crash recovery occurs at instance startup, this parameter is useful for specifying the number of processes to use for crash recovery. The value of this parameter is also the default number of processes used for media recovery if you do not specify the `PARALLEL` clause of the `RECOVER` statement. To use parallel processing, the value of `RECOVERY_PARALLELISM` must be greater than 1 and cannot exceed the value of the `PARALLEL_MAX_SERVERS` parameter. Parallel block recovery requires a minimum of eight recovery processes for it to be more effective than serial recovery.

Recovery is usually I/O bound on reads to data blocks. Consequently, parallelism at the block level may only help recovery performance if it increases total I/Os. In other words, parallelism at the block level by-passes operating system restrictions on asynchronous I/Os. Performance on systems with efficient asynchronous I/O typically does not improve significantly with parallel block recovery.

Tuning the Rolling Back Phase

During the second phase of instance recovery, Oracle rolls back uncommitted transactions. Oracle uses two features, fast-start on-demand rollback and fast-start parallel rollback, to increase the efficiency of this recovery phase.

Note: These features are part of fast-start fault recovery and are only available in the Oracle8*i* Enterprise Edition.

This section contains the following topics:

- Using Fast-Start On-Demand Rollback
- Using Fast-Start Parallel Rollback

Using Fast-Start On-Demand Rollback

Using the fast-start on-demand rollback feature, Oracle automatically allows new transactions to begin immediately after the roll forward phase of recovery completes. Should a user attempt to access a row that is locked by a dead transaction, Oracle rolls back only those changes necessary to complete the transaction, in other words, it rolls them back *on demand*. Consequently, new transactions do not have to wait until all parts of a long transaction are rolled back.

Note: Oracle does this automatically. You do not need to set any parameters or issue statements to use this feature.

Using Fast-Start Parallel Rollback

In fast-start parallel rollback, the background process SMON acts as a coordinator and rolls back a set of transactions in parallel using multiple server processes. Essentially, fast-start parallel rollback is to rolling back what parallel block recovery is to rolling forward.

Fast-start parallel rollback is mainly useful when a system has transactions that run a long time before committing, especially parallel `INSERT`, `UPDATE`, and `DELETE` operations. When SMON discovers that the amount of recovery work is above a certain threshold, it automatically begins parallel rollback by dispersing the work among several parallel processes: process 1 rolls back one transaction, process 2 rolls back a second transaction, and so on. The threshold is the point at which parallel recovery becomes cost-effective, in other words, when parallel recovery takes less time than serial recovery.

One special form of fast-start parallel rollback is intra-transaction recovery. In intra-transaction recovery, a single transaction is divided among several processes. For example, assume 8 transactions require recovery with one parallel process assigned to each transaction. The transactions are all similar in size except for transaction 5, which is quite large. This means it takes longer for one process to roll this transaction back than for the other processes to roll back their transactions.

In this situation, Oracle automatically begins intra-transaction recovery by dispersing transaction 5 among the processes: process 1 takes one part, process 2 takes another part, and so on.

You control the number of processes involved in transaction recovery by setting the parameter `FAST_START_PARALLEL_ROLLBACK` to one of three values:

`FALSE`	Turns off fast-start parallel rollback.
`LOW`	Specifies that the number of recovery servers may not exceed twice the value of the `CPU_COUNT` parameter.
`HIGH`	Specifies that the number of recovery servers may not exceed four times the value of the `CPU_COUNT` parameter.

Parallel Rollback in an Oracle Parallel Server Configuration In Oracle Parallel Server, you can perform fast-start parallel rollback on each instance. Within each instance, you can perform parallel rollback on transactions that are:

- Online on a given instance.
- Offline and not being recovered on instances other than the given instance.

After a rollback segment is online for a given instance, only this instance can perform parallel rollback on transactions on that segment.

Monitoring Progress of Fast-Start Parallel Rollback Monitor the progress of fast-start parallel rollback by examining the `V$FAST_START_SERVERS` and `V$FAST_START_TRANSACTIONS` tables. `V$FAST_START_SERVERS` provides information about all recovery processes performing fast-start parallel rollback. `V$FAST_START_TRANSACTIONS` contains data about the progress of the transactions.

> **See Also:** For more information on fast-start parallel rollback in an Oracle Parallel Server environment, see *Oracle8i Parallel Server Administration, Deployment, and Performance.* For more information about initialization parameters, see the *Oracle8i Reference.*

Index

A

ABORTED_REQUEST_THRESHOLD
procedure, 19-26
access methods, 4-20
cluster scans, 4-22
execution plans, 4-5
hash scans, 4-22
index scans, 4-22
table scans, 4-21
access path, 2-11
access paths
cluster join, 4-38
composite index, 4-39
defined, 4-7
hash cluster key, 4-38
indexed cluster key, 4-39
optimization, 4-20
single row by cluster join, 4-36
single row by hash cluster key (with unique key), 4-36
single row by rowid, 4-35
single row by unique or primary key, 4-37
alert files, 11-4
ALL, 4-67
ALL_INDEXES view, 12-17
ALL_OBJECTS view, 19-34
ALL_ROWS hint, 4-11, 7-6
allocation
of memory, 19-2
ALTER INDEX REBUILD statement, 12-9
ALTER SESSION statement
examples, 6-5
HASH_JOIN_ENABLED, 4-53
OPTIMIZER_GOAL, 4-11
SET SESSION_CACHED_CURSORS statement, 19-21
ALTER SYSTEM statement
CHECKPOINT clause, 24-7
MTS_DISPATCHERS parameter, 21-8
ALWAYS_ANTI_JOIN parameter, 4-29, 4-61, 7-23, 7-24
ALWAYS_SEMI_JOIN parameter, 4-62
analysis dictionary, 11-5
ANALYZE statement, 20-31
creating histograms, 8-18
AND_EQUAL hint, 7-16, 12-7
anti-joins, 4-61
ANY, 4-67
APPEND hint, 7-28
application design, 2-9
application designer, 1-8
application developer, 1-8
applications
client/server, 3-11
data warehousing
star queries, 4-62
decision support, 3-4, 9-15
distributed databases, 3-8
OLTP, 3-2
Oracle Parallel Server, 3-10
parallel query, 3-5
registering with the database, 11-8
ARCH process, 19-8
architecture and CPU, 18-13
array interface, 22-16
audit trail, 11-4
Average Elapsed Time data view, 14-11

B

B*-tree index, 12-16, 12-20
backups
  cumulative incremental, 20-50, 20-51, 20-52, 20-53
  tuning, 20-61
bandwidth, 9-15
Basic Statistics for Parse/Execute/Fetch drilldown data view, 14-19
BEGIN_DISCRETE_TRANSACTION procedure, 17-2, 17-3
benefits
  of tuning, 2-3
BETWEEN, 4-68
binary files
  formatting using Oracle Trace, 14-5
bind variables, 19-17
  optimization, 4-27
BITMAP CONVERSION row source, 12-20
bitmap indexes, 12-14, 12-19
  creating, 12-16
  inlist iterator, 5-22
  maintenance, 12-15
  scans of, 4-24
  size, 12-21
  storage considerations, 12-15
  when to use, 12-13
BITMAP keyword, 12-16
BITMAP_MERGE_AREA_SIZE parameter, 4-30, 12-16, 12-19
bitmaps
  mapping to rowids, 12-18
block contention, 2-13
block sampling, 8-3
blocks, 20-10
bottlenecks
  disk I/O, 20-18
  memory, 19-2
broadcast
  distribution value, 5-8
buffer caches, 2-12
  memory allocation, 19-31
  partitioning, 19-35
  reducing buffers, 19-31
  reducing cache misses, 19-31
  tuning, 19-27
buffer get, 2-10
buffer not pinned statistics, 19-29
buffer pinned statistics, 19-29
buffer pools
  default cache, 19-33
  keep cache, 19-33
  multiple, 19-32, 19-33
  RECYCLE cache, 19-33
  syntax, 19-36
BUFFER_POOL clause, 19-36
BUFFER_POOL_name parameter, 19-35
business rules, 1-8, 2-3, 2-7
BYTES column
  PLAN_TABLE table, 5-5

C

CACHE hint, 7-30
cache hit ratios
  increasing, 19-31
cardinality, 12-21
CARDINALITY column
  PLAN_TABLE table, 5-5
Cartesian products, 4-48
CATPARR.SQL script, 19-31
CATPERF.SQL file, 19-38
chained rows, 20-31
channel bandwidth, 16-6
CHECKPOINT clause
  ALTER SYSTEM statement, 24-7
checkpoints
  choosing checkpoint frequency, 20-40
CHOOSE hint, 4-11, 7-8
client/server applications, 3-11, 18-5
CLUSTER hint, 7-11
cluster joins, 4-55
clusters, 12-25
  hash
    scans of, 4-22, 4-36, 4-38
  index
    scans of, 4-39
  joins and, 4-36, 4-38, 4-55
  scans of, 4-22, 4-36

hash, 4-36, 4-38
joins, 4-38
collections, 14-4, 14-23
columns
pseudocolumns
ROWNUM, 4-46, 4-77, 4-88
selectivity, 8-2
histograms, 8-17
to index, 12-4
COMPATIBLE parameter
and parallel query, 7-24
complex view merging, 4-78
composite indexes, 12-5
composite partitions
examples of, 5-16
CONNECT BY clause
optimizing view queries, 4-77
connection manager, 22-17
connection pooling, 21-8
consistency
read, 18-10
consistent gets statistic, 19-28, 21-4, 21-21
consistent mode
TKPROF, 6-13
constants
comparisons and, 4-65
evaluation of expressions, 4-65
when computed, 4-65
constraints, 12-11
contention
disk access, 20-18
free lists, 21-20
memory, 19-2
memory access, 21-1
redo allocation latch, 21-19
redo copy latches, 21-19
rollback segments, 21-3
tuning, 21-1
tuning resource, 2-13
context area, 2-12
context switching, 18-6
COST column
PLAN_TABLE table, 5-5
cost-based optimization, 4-12
extensible optimization, 4-32
histograms, 8-17
procedures for plan stability, 10-8
selectivity of predicates, 8-2
histograms, 8-17
user-defined, 4-33
star queries, 4-62
statistics, 4-10, 8-2
user-defined, 4-33
upgrading to, 10-9
user-defined costs, 4-33
count column
SQL trace, 6-12
CPU
checking utilization, 18-4
column
SQL trace, 6-12
detecting problems, 18-4
system architecture, 18-13
tuning, 18-1
utilization, 9-15
CPU Statistics data view, 14-12
CPU Statistics for Parse/Execute/Fetch drilldown data view, 14-19
CPU_COUNT initialization parameter, 24-17
CREATE CLUSTER statement, 12-27
CREATE INDEX statement
examples, 20-38
NOSORT clause, 20-38
CREATE OUTLINE statement, 10-5
CREATE TABLE statement
STORAGE clause, 20-24
TABLESPACE clause, 20-24
CREATE TABLESPACE statement, 20-23
CREATE_BITMAP_AREA_SIZE parameter, 12-16, 12-19
CREATE_STORED_OUTLINES parameter, 10-4
cross joins, 4-48
current column
SQL trace, 6-13
current mode
TKPROF, 6-13
CURSOR_NUM column
TKPROF_TABLE table, 6-18
CURSOR_SHARING parameter, 19-18

CURSOR_SPACE_FOR_TIME parameter
setting, 19-19

D

data
comparative, 11-5
design tuning, 2-8
sources for tuning, 11-2
volume, 11-2
data blocks, 20-10
data cache, 23-2
data dictionary, 2-12, 11-3, 19-22
statistics in, 4-10, 8-13
views used in optimization, 8-13
data views in Oracle Trace, 14-6
Average Elapsed Time, 14-11
CPU Statistics, 14-12
Disk Reads, 14-9
Disk Reads/Execution Ratio, 14-9
Disk Reads/Logical Reads Ratio, 14-10
Disk Reads/Rows Fetched Ratio, 14-9
Execute Elapsed Time, 14-11
Fetch Elapsed Time, 14-11
Logical Reads, 14-9
Logical Reads/Rows Fetched Ratio, 14-9
Number of Rows Processed, 14-12
Parse Elapsed Time, 14-11
Parse/Execution Ratio, 14-10
Re-Parse Frequency, 14-10
Rows Fetched/Fetch Count Ratio, 14-12
Rows Sorted, 14-12
Sorts in Memory, 14-12
Sorts on Disk, 14-12
Total Elapsed Time, 14-11
Waits by Average Wait Time, 14-13
Waits by Event Frequency, 14-13
Waits by Total Wait Time, 14-13
data warehousing
dimensions, 4-62
star queries, 4-62
database
buffers, 19-31
database administrator (DBA), 1-8
Database Connection event, 14-5
Database Resource Manager, 18-4, 18-7, 23-4, 23-5
database writer process (DBWn)
tuning, 18-10
databases
distributed
statement optimization on, 4-94
DATAFILE clause, 20-23
datafiles
placement on disk, 20-19
datatypes
user-defined
statistics, 4-33
DATE_OF_INSERT column
TKPROF_TABLE table, 6-18
db block gets statistic, 19-28, 21-4, 21-21
DB_BLOCK_BUFFERS parameter, 19-31, 19-36, 20-45
DB_BLOCK_LRU_LATCHES parameter, 19-36, 19-40
DB_BLOCK_SIZE parameter
tuning backups, 20-63
DB_FILE_DIRECT_IO_COUNT parameter, 20-63
DB_FILE_MULTIBLOCK_READ_COUNT parameter, 4-26, 4-29, 20-38
cost-based optimization, 4-59
DB_WRITER_PROCESSES initialization parameter, 20-43, 20-44
DBA_INDEXES view, 12-17
DBA_OBJECTS view, 19-34
DBMS_APPLICATION_INFO package, 3-7
DBMS_SHARED_POOL package, 13-4, 19-13, 19-26
DBMS_STATS package, 8-5, 8-6
creating histograms, 8-18
DBMSPOOL.SQL script, 13-4, 19-13
decision support, 3-4
systems (DSS), 1-2
tuning, 9-15
with OLTP, 3-6
decomposition of SQL statements, 9-32
default cache, 19-33
demand rate, 1-5
DEPTH column
TKPROF_TABLE table, 6-18

design dictionary, 11-5
detail report in Oracle Trace, 14-6
details property sheet in Oracle Trace, 14-15
DETERMINISTIC functions, 4-70
deterministic functions, 4-70
device bandwidth, 16-6
  evaluating, 20-11
device latency, 16-6
diagnosing tuning problems, 16-1
dictionary-mapped tablespaces, 20-29
dimensions
  star joins, 4-62
  star queries, 4-62
disabled constraints, 12-11
discrete transactions
  example, 17-4
  processing, 17-3
  when to use, 17-2
disk column
  SQL trace, 6-12
Disk Reads data view, 14-9
Disk Reads/Execution Ratio data view, 14-9
Disk Reads/Logical Reads Ratio data view, 14-10
Disk Reads/Rows Fetched Ratio data view, 14-9
DISKRATIO parameter
  to distribute backup I/O, 20-61
disks
  contention, 20-18, 20-19
  distributing I/O, 20-19
  I/O requirements, 20-4
  layout options, 20-10
  monitoring OS file activity, 20-15
  number required, 20-4
  placement of datafiles, 20-19
  placement of redo logs, 20-19
  reducing contention, 20-18
  speed characteristics, 20-3
  testing performance, 20-5
dispatcher processes (Dnnn), 21-8
DISTINCT operator
  optimizing views, 4-78
distributed databases, 3-8
  statement optimization on, 4-94
distributed query, 9-30, 9-40
distributed transactions
  distributed statements, 4-48
  optimizing, 4-94
  sample table scan not supported, 4-21
distributing I/O, 20-19, 20-23
distribution
  hints for, 7-26
DISTRIBUTION column
  PLAN_TABLE table, 5-6
DIUTIL package, 13-4
domain indexes
  and EXPLAIN PLAN, 5-22
  extensible optimization, 4-32
  user-defined statistics, 4-33
  using, 12-24
drilldown data views in Oracle Trace, 14-17
  Basic Statistics for Parse/Execute/Fetch, 14-19
  CPU Statistics for Parse/Execute/Fetch, 14-19
  Parse Statistics, 14-19
  Row Statistics for Execute/Fetch, 14-20
duration events in Oracle Trace, 14-5
dynamic extension, 20-27
  avoiding, 20-29
dynamic performance views
  enabling statistics, 6-4
  for tuning, 15-1

E

elapsed column
  SQL trace, 6-12
enabled constraints, 12-11
enforced constraints, 12-11
equijoins, 9-5
  cluster joins, 4-55
  defined, 4-47
  hash joins, 4-53
  sort-merge, 4-51
errors
  common tuning, 2-15
  during discrete transactions, 17-3
events in Oracle Trace, 14-5
examples
  ALTER SESSION statement, 6-5
  CREATE INDEX statement, 20-38
  DATAFILE clause, 20-23

discrete transactions, 17-4
execution plan, 9-3
EXPLAIN PLAN output, 6-15, 9-3
full table scan, 9-3
indexed query, 9-4
NOSORT clause, 20-38
SET TRANSACTION statement, 20-30
SQL trace facility output, 6-15
STORAGE clause, 20-24
table striping, 20-23
TABLESPACE clause, 20-24
executable code as data source, 11-4
Execute Elapsed Time data view, 14-11
execution plan
accessing views, 4-80, 4-82, 4-84
complex statements, 4-75
compound queries, 4-91, 4-92, 4-93
joining views, 4-89
joins, 4-49, 4-58
OR operators, 4-73
execution plans, 5-2
examples, 4-75, 6-7, 9-3
execution sequence of, 4-7
overview of, 4-5
plan stability, 10-2
preserving with plan stability, 10-2
TKPROF, 6-7, 6-10
viewing, 4-3
expectations for tuning, 1-9
EXPLAIN PLAN statement
access paths, 4-21, 4-24, 4-35, 4-36, 4-37, 4-38, 4-39, 4-40, 4-41, 4-42, 4-43, 4-44, 4-45, 4-47
and domain indexes, 5-22
and full partition-wise joins, 5-20
and partial partition-wise joins, 5-19
and partitioned objects, 5-14
examples of output, 6-15, 9-3
introduction, 11-6
invoking with the TKPROF program, 6-10
PLAN_TABLE table, 5-3
restrictions, 5-23
SQL decomposition, 9-35
Export utility
copying statistics, 8-2
extensible optimization, 4-32
user-defined costs, 4-33
user-defined selectivity, 4-33
user-defined statistics, 4-33
extents
unlimited, 20-28

F

fact tables
star joins, 4-62
star queries, 4-62
fast full index scans, 4-23, 12-8
FAST_START_IO_TARGET initialization parameter
controlling checkpoints with, 20-40
recovery time and the, 24-5
FAST_START_PARALLEL_ROLLBACK initialization parameter, 24-17
fast-start checkpoints
controlling checkpoints, 20-40
FAST_START_IO_TARGET initialization parameter, 24-5
LOG_CHECKPOINT_INTERVAL initialization parameter, 24-5
LOG_CHECKPOINT_TIMEOUT initialization parameter, 24-4
fast-start on-demand rollback, 24-16
fast-start parallel rollback, 24-16
Fetch Elapsed Time data view, 14-11
file storage
designing, 20-5
FILESPERSET parameter
tuning backups, 20-63
FIRST_ROWS hint, 4-11, 7-7
FORMAT statement
in Oracle Trace, 14-21
formatter tables
in Oracle Trace, 14-5
free lists
adding, 21-21
contention, 21-20
reducing contention, 21-21
FULL hint, 7-10, 12-7
full index scans, 4-23
full partition-wise joins, 5-20
full table scans, 4-21, 4-46, 9-3

multiblock reads, 4-26
rule-based optimizer, 4-46
selectivity and, 4-25
function-based indexes, 12-12
functions
PL/SQL
DETERMINISTIC, 4-70
deterministic, 4-70
SQL
optimizing view queries, 4-84
user-defined
extensible optimization, 4-32

G

GATHER_ INDEX_STATS procedure
in DBMS_STATS package, 8-6
GATHER_DATABASE_STATS procedure
in DBMS_STATS package, 8-6
GATHER_SCHEMA_STATS procedure
in DBMS_STATS package, 8-6
GATHER_TABLE_STATS procedure
in DBMS_STATS package, 8-6
GETMISSES column
in V$ROWCACHE table, 19-22
GETS column
in V$ROWCACHE table, 19-22
global hints, 7-37
goals for tuning, 1-9, 2-14
GROUP BY clause
NOSORT clause, 20-39
optimizing views, 4-78

H

hash
distribution value, 5-8
hash areas, 2-12
hash clusters
scans of, 4-22, 4-36, 4-38
HASH hint, 7-11
hash join, 4-53
HASH_AREA_SIZE parameter, 4-54
HASH_MULTIBLOCK_IO_COUNT
parameter, 4-54
index join, 4-24
hash partitions, 5-14
examples of, 5-14
HASH_AJ hint, 4-61, 7-22, 7-23
HASH_AREA_SIZE parameter, 4-29, 4-54
HASH_JOIN_ENABLED parameter, 4-29, 4-53
HASH_MULTIBLOCK_IO_COUNT
parameter, 4-30, 4-54
HASH_SJ hint, 4-62, 7-23
hashing, 12-26
HASHKEYS parameter
CREATE CLUSTER statement, 12-27
HIGH_VALUE statistics, 4-26
hints, 7-2
access methods, 7-9
ALL_ROWS hint, 7-6
AND_EQUAL hint, 7-16, 12-7
as used in outlines, 10-3
CACHE hint, 7-30
cannot override sample access path, 4-25
CHOOSE hint, 7-8
CLUSTER hint, 7-11
degree of parallelism, 7-24
extensible optimization, 4-33
FIRST_ROWS hint, 7-7
FULL hint, 7-10, 12-7
global, 7-37
HASH hint, 7-11
HASH_AJ hint, 7-22
HASH_SJ hint, 7-23
how to use, 7-2
INDEX hint, 7-12, 7-19, 12-7
INDEX_ASC hint, 7-13
INDEX_DESC hint, 7-14, 7-15
INDEX_FFS, 4-23
INDEX_FFS hint, 7-15
INDEX_JOIN, 4-24
join operations, 7-19
LEADING hint, 7-22
MERGE_AJ and HASH_AJ, 4-61
MERGE_AJ hint, 7-22
MERGE_SJ and HASH_SJ, 4-62
MERGE_SJ hint, 7-23
NO_EXPAND hint, 7-17
NO_INDEX, 12-7

NO_INDEX hint, 7-15
NO_MERGE hint, 7-32
NO_PUSH_PRED hint, 7-34
NO_UNNEST hint, 7-33
NOCACHE hint, 7-31
NOPARALLEL hint, 7-26
NOREWRITE hint, 7-18
optimization approach and goal, 7-6
ORDERED, 4-59
ORDERED hint, 7-18
overriding optimizer choice, 4-25
overriding OPTIMIZER_MODE and OPTIMIZER_GOAL, 4-11
PARALLEL hint, 7-25
parallel query option, 7-24
PQ_DISTRIBUTE hint, 7-26
PUSH_JOIN_PRED, 4-87
PUSH_PRED hint, 7-34
PUSH_SUBQ hint, 7-34
REWRITE hint, 7-17
ROWID hint, 7-11
RULE hint, 7-8
STAR hint, 7-19
UNNEST hint, 7-32
USE_CONCAT hint, 7-16
USE_HASH, 4-53
USE_MERGE hint, 7-21
USE_NL hint, 7-20
histograms, 8-17
number of buckets, 8-19
HOLD_CURSOR clause, 19-10

I

ID column
PLAN_TABLE table, 5-5
Import utility
copying statistics, 8-2
IN operator, 4-66
merging views, 4-79
IN subquery, 4-78
INDEX hint, 7-12, 12-7, 12-17
index joins, 4-24
INDEX_ASC hint, 7-13
INDEX_COMBINE hint, 12-7, 12-17
INDEX_DESC hint, 7-14, 7-15
INDEX_FFS hint, 4-23, 7-15, 12-9
INDEX_JOIN hint, 4-24
indexes
avoiding the use of, 12-7
bitmap, 12-13, 12-14, 12-16, 12-19
choosing columns for, 12-4
cluster
scans of, 4-39
composite, 12-5
scans of, 4-39
design, 2-9
domain, 12-24
domain indexes
extensible optimization, 4-32
user-defined statistics, 4-33
enforcing uniqueness, 12-10
ensuring the use of, 12-6
example, 9-4
fast full scan, 12-8
fast full scans of, 4-23
function-based, 12-12
index joins, 4-24
modifying values of, 12-5
non-unique, 12-10
optimization and, 4-71
placement on disk, 20-21
range scans, 4-23
rebuilding, 12-9
recreating, 12-9
scans of, 4-22
bounded range, 4-41
cluster key, 4-39
composite, 4-39
MAX or MIN, 4-44
ORDER BY, 4-45
restrictions, 4-46
single-column, 4-40
unbounded range, 4-42
selectivity of, 12-4
statement conversion and, 4-71
statistics, gathering, 8-6
unique scans, 4-22
when to create, 12-2

initialization parameters
ALWAYS_ANTI_JOIN, 4-61
ALWAYS_SEMI_JOIN, 4-62
CPU_COUNT, 24-17
DB_FILE_MULTIBLOCK_READ_COUNT, 4-26, 4-59
FAST_START_PARALLEL_ROLLBACK, 24-17
HASH_AREA_SIZE, 4-54
HASH_JOIN_ENABLED, 4-53
HASH_MULTIBLOCK_IO_COUNT, 4-54
in Oracle Trace, 14-22
LOG_CHECKPOINT_INTERVAL, 24-5
LOG_CHECKPOINT_TIMEOUT, 24-4
MAX_DUMP_FILE_SIZE, 6-4
OPTIMIZER_FEATURES_ENABLE, 4-23, 4-24, 4-78, 4-87
OPTIMIZER_MODE, 4-10, 4-34, 7-6
OPTIMIZER_PERCENT_PARALLEL, 4-9
PARALLEL_MAX_SERVERS, 24-15
PRE_PAGE_SGA, 19-5
RECOVERY_PARALLELISM, 24-15
SESSION_CACHED_CURSORS, 19-20
SORT_AREA_SIZE, 4-59
SQL_TRACE, 6-5
TIMED_STATISTICS, 6-4
USER_DUMP_DEST, 6-4
IN-lists, 7-13, 7-17
INSERT statement
append, 7-28
internal write batch size, 20-45
INTERSECT operator
compound queries, 4-48
example, 4-93
optimizing view queries, 4-77
intra transaction recovery, 24-17
I/O
analyzing needs, 20-2, 20-3
balancing, 20-22
distributing, 20-19, 20-23
insufficient, 16-6
multiple buffer pools, 19-33
parallel execution, 9-15
Statistics for Parse/Execute/Fetch view, 14-19
testing disk performance, 20-5
tuning, 2-12, 20-2
isolation level
of transactions, 17-6

J

joins
anti-joins, 4-61
Cartesian products, 4-48
cluster, 4-36, 4-55
searches on, 4-38
convert to subqueries, 4-74
cross, 4-48
defined, 4-47
equijoins, 4-47
execution plans and, 4-49
hash joins, 4-53
index joins, 4-24
join order
execution plans, 4-5
selectivity of predicates, 4-33, 8-2, 8-17
nested loops, 4-50
cost-based optimization, 4-59
non-equijoins, 4-47
optimization of, 4-60
outer, 4-47
non-null values for nulls, 4-86
parallel, and PQ_DISTRIBUTE hint, 7-26
partition-wise
examples of full, 5-20
examples of partial, 5-19
full, 5-20
sample table scan not supported, 4-21
select-project-join views, 4-76
semi-joins, 4-61
sort-merge, 4-51
cost-based optimization, 4-59
example, 4-43
star joins, 4-62
star queries, 4-62

K

keep cache, 19-33
keys
searches, 4-36

L

large pool, 20-64
LARGE_POOL_SIZE parameter, 20-64
latches
  contention, 2-13, 18-12
  redo allocation latch, 21-16
  redo copy latches, 21-16
LEADING hint, 7-22
least recently used list (LRU), 18-10
library cache, 2-12
  memory allocation, 19-16
  tuning, 19-14
LIKE, 4-66
load balancing, 20-22
lock contention, 2-13
log, 21-16
log buffer tuning, 2-12, 19-7
log writer process (LGWR) tuning, 20-19, 20-41
LOG_BUFFER parameter, 19-6, 20-41
  setting, 19-8
LOG_CHECKPOINT_INTERVAL initialization parameter, 20-40
  recovery time, 24-5
LOG_CHECKPOINT_TIMEOUT initialization parameter, 20-40
  recovery time, 24-4
LOG_SIMULTANEOUS_COPIES parameter, 21-19
LOG_SMALL_ENTRY_MAX_SIZE parameter, 21-19
Logical Reads data view, 14-9
Logical Reads/Rows Fetched Ratio data view, 14-9
logical structure of database, 2-9
long waits
  definition of, 20-57
lookup tables
  star queries, 4-62
LOW_VALUE statistics, 4-26
LRU
  aging policy, 19-33
  latch, 19-35, 19-36, 19-41
  latch contention, 19-41, 21-19

M

Management Information Base (MIB), 11-5
massively parallel system, 9-15
max session memory statistic, 19-24
MAX_DUMP_FILE_SIZE
  SQL Trace parameter, 6-4
MAX_DUMP_FILE_SIZE initialization parameter, 6-4
MAXOPENCURSORS clause, 19-10
MAXOPENFILES parameter
  tuning backups, 20-63
memory
  insufficient, 16-5
  reducing usage, 19-42
  tuning, 2-11
memory allocation
  buffer caches, 19-31
  importance, 19-2
  library cache, 19-16
  shared SQL areas, 19-16
  sort areas, 20-35
  tuning, 19-2, 19-42
  users, 19-6
MERGE hint, 7-31
MERGE_AJ hint, 4-61, 7-22, 7-23
MERGE_SJ hint, 4-62, 7-23
merging complex views, 4-78
merging views into statements, 4-76
message rate, 16-7
method
  applying, 2-14
  tuning, 2-1
  tuning steps, 2-5
MIB, 11-5
migrated rows, 20-31
MINUS operator
  compound queries, 4-48
  optimizing view queries, 4-77
mirroring
  redo logs, 20-21
monitoring, 11-5
MTS_DISPATCHERS parameter, 21-8
MTS_MAX_DISPATCHERS parameter, 21-8
MTS_MAX_SERVERS parameter, 21-11

multiblock reads, 20-28
multiple buffer pools, 19-32, 19-33, 19-36
multi-purpose applications, 3-6
multi-threaded server
  context area size, 2-12
  performance issues, 21-5
  reducing contention, 21-5
  tuning, 21-5
  tuning memory, 19-22
multi-tier systems, 3-9, 18-15

N

NAMESPACE column
  V$LIBRARYCACHE table, 19-14
nested loops joins, 4-50
  cost-based optimization, 4-59
Net8 Assistant, 22-17
network
  array interface, 22-16
  bandwidth, 16-6
  constraints, 16-6
  detecting performance problems, 22-9
  prestarting processes, 22-6
  problem solving, 22-11
  Session Data Unit, 22-16
  tuning, 22-1
NLS_SORT parameter
  ORDER BY access path, 4-45
NO_EXPAND hint, 7-17
NO_INDEX hint, 7-15, 12-7
NO_MERGE hint, 7-32
NO_PUSH_PRED hint, 7-34
NO_UNNEST hint, 7-33
NOAPPEND hint, 7-28
NOCACHE hint, 7-31
non-equijoins
  defined, 4-47
NOPARALLEL hint, 7-26
NOPARALLEL_INDEX hint, 7-30
NOREWRITE hint, 7-18
NOSORT clause, 20-38, 20-39
NOT, 4-68
NOT IN subquery, 4-61
NT performance, 23-6
nulls
  converting to values
    optimization, 4-86
  non-null values for, 4-86
NUM_DISTINCT column
  USER_TAB_COLUMNS view, 4-26
NUM_ROWS column
  USER_TABLES view, 4-26
Number of Rows Processed data view, 14-12

O

OBJECT_INSTANCE column
  PLAN_TABLE table, 5-5
OBJECT_NAME column
  PLAN_TABLE table, 5-5
OBJECT_NODE column
  PLAN_TABLE table, 5-4
OBJECT_OWNER column
  PLAN_TABLE table, 5-5
OBJECT_TYPE column
  PLAN_TABLE table, 5-5
online transaction processing (OLTP), 1-2, 3-2
  with decision support, 3-6
OPEN_CURSORS parameter
  allocating more private SQL areas, 19-10
  increasing cursors per session, 19-16
operating system
  data cache, 23-2
  monitoring disk I/O, 20-15
  monitoring tools, 11-3
  tuning, 2-13, 16-7, 19-4
OPERATION column
  PLAN_TABLE table, 5-4, 5-9
OPTIMAL storage parameter, 20-30
optimization
  choosing the approach, 4-10
  conversion of expressions and predicates, 4-65
  cost-based, 4-12, 4-59
    choosing an access path, 4-25
    examples of, 4-26
    histograms, 8-17
    remote databases and, 4-94
    star queries, 4-62
    user-defined costs, 4-33

described, 4-4
DISTINCT, 4-78
distributed SQL statements, 4-94
extensible optimizer, 4-32
GROUP BY views, 4-78
hints, 4-11, 4-23, 4-24
manual, 4-11
merging complex views, 4-78
merging views into statements, 4-76
non-null values for nulls, 4-86
operations performed, 4-48
rule-based, 4-34, 4-60
selectivity of predicates, 8-2
histograms, 8-17
user-defined, 4-33
selectivity of queries and, 4-25
select-project-join views, 4-76
semi-joins, 4-61
statistics, 4-10, 8-2
user-defined, 4-33
transitivity and, 4-69
types of SQL statements, 4-47
without merging, 4-88
optimizer, 4-4
plan stability, 10-2
OPTIMIZER column
PLAN_TABLE, 5-5
OPTIMIZER_FEATURES_ENABLE
parameter, 4-23, 4-24, 4-78, 4-87
OPTIMIZER_FEATURES_ENABLED
parameter, 4-29
OPTIMIZER_GOAL clause, 4-11
OPTIMIZER_INDEX_CACHING, 4-31
OPTIMIZER_INDEX_COST_ADJ parameter, 4-30
OPTIMIZER_MODE, 4-10, 4-20
hints affecting, 4-11
OPTIMIZER_MODE initialization parameter, 4-12, 4-29, 4-34, 7-6
OPTIMIZER_PERCENT_PARALLEL initialization parameter, 4-9
OPTIMIZER_PERCENT_PARALLEL
parameter, 4-9, 4-29
OPTIONS column
PLAN_TABLE table, 5-4
Oracle Enterprise Manager, 11-8
Oracle Expert, 2-1, 11-13
Oracle Forms, 6-5
control of parsing and private SQL areas, 19-11
Oracle Parallel Server, 3-10
CPU, 18-17
synchronization points, 2-8
tuning, 11-14
Oracle Parallel Server Management, 11-14
Oracle Performance Manager, 11-11
Oracle Server
client/server configuration, 3-11
configurations, 3-7
events, 14-5
Oracle Trace, 14-1, 19-40
accessing collected data, 14-5
binary files, 14-5
collection results, 14-27
collections, 14-4, 14-23
command-line interface, 14-20
data views, 14-6
Average Elapsed Time, 14-11
CPU Statistics, 14-12
Disk Reads, 14-9
Disk Reads/Execution Ratio, 14-9
Disk Reads/Logical Reads Ratio, 14-10
Disk Reads/Rows Fetched Ratio, 14-9
Execute Elapsed Time, 14-11
Fetch Elapsed Time, 14-11
Logical Reads, 14-9
Logical Reads/Rows Fetched Ratio, 14-9
Number of Rows Processed, 14-12
Parse Elapsed Time, 14-11
Parse/Execution Ratio, 14-10
Re-Parse Frequency, 14-10
Rows Fetched/Fetch Count Ratio, 14-12
Rows Sorted, 14-12
Sorts in Memory, 14-12
Sorts on Disk, 14-12
Total Elapsed Time, 14-11
Waits by Average Wait Time, 14-13
Waits by Event Frequency, 14-13
Waits by Total Wait Time, 14-13
deleting files, 14-22
details property sheet, 14-15
drilldown data views, 14-17, 14-19

Basic Statistics for Parse/Execute/Fetch view, 14-19
CPU Statistics for Parse/Execute/Fetch view, 14-19
Parse Statistics view, 14-19
Row Statistics for Execute/Fetch view, 14-20
duration events, 14-5
events, 14-5
FORMAT statement, 14-21
formatter tables, 14-5
formatting data, 14-27
Oracle Trace Data Viewer, 14-6
parameters, 14-22
point events, 14-5
predefined data views, 14-6
reporting utility, 14-6, 14-28
SQL statement property sheet, 14-15
START statement, 14-21
STOP statement, 14-21
stored procedures, 14-25
using to collect workload data, 14-3
viewing data, 14-13
Oracle Trace Data Viewer, 14-6
Oracle Trace Manager, 14-4, 14-23
used for formatting collections, 14-5
ORACLE_TRACE_COLLECTION_NAME parameter, 14-22, 14-23
ORACLE_TRACE_COLLECTION_PATH parameter, 14-22
ORACLE_TRACE_COLLECTION_SIZE parameter, 14-22
ORACLE_TRACE_ENABLE parameter, 14-22, 14-23
ORACLE_TRACE_FACILITY_NAME parameter, 14-22, 14-23
ORACLE_TRACE_FACILITY_PATH parameter, 14-22
ORDERED hint, 4-59, 7-18
ORDERED_PREDICATES hint, 7-35
OTHER column
PLAN_TABLE table, 5-6
OTHER_TAG column
PLAN_TABLE table, 5-5
outer joins
defined, 4-47
non-null values for nulls, 4-86
outlines
CREATE OUTLINE statement, 10-5
creating and using, 10-4
execution plans and plan stability, 10-2
hints, 10-3
matching with SQL statements, 10-3
moving tables, 10-7
storage requirements, 10-4
using, 10-5
using to move to the cost-based optimizer, 10-8
viewing data for, 10-6
overloaded disks, 20-19

P

packages
DBMS_APPLICATION_INFO package, 3-7
DBMS_SHARED_POOL package, 13-4
DBMS_TRANSACTION package, 17-4
DIUTIL package, 13-4
registering with the database, 11-8
STANDARD package, 13-4
page table, 18-5
paging, 16-5, 18-5
library cache, 19-16
reducing, 19-4
SGA, 19-42
PARALLEL clause
RECOVER statement, 24-15
parallel execution, 3-5
hints, 7-25
query servers, 21-15
tuning query servers, 21-15
PARALLEL hint, 7-25
parallel joins
and PQ_DISTRIBUTE hint, 7-26
parallel recovery, 24-15
PARALLEL_MAX_SERVERS initialization parameter, 24-15
PARALLEL_MAX_SERVERS parameter, 24-15
parameter files, 11-4
PARENT_ID column
PLAN_TABLE table, 5-5
Parse Elapsed Time data view, 14-11

Parse Statistics drilldown data view, 14-19
Parse/Execution Ratio data view, 14-10
parsing
  Oracle Forms, 19-11
  Oracle precompilers, 19-10
  reducing unnecessary calls, 19-10
partition views, 9-35
PARTITION_ID column
  PLAN_TABLE table, 5-6
PARTITION_START column
  PLAN_TABLE table, 5-6
PARTITION_STOP column
  PLAN_TABLE table, 5-6
PARTITION_VIEW_ENABLED parameter, 9-36
partitioned objects
  and EXPLAIN PLAN statement, 5-14
partitioning
  distribution value, 5-8
  examples of, 5-14
  examples of composite, 5-16
  hash, 5-14
  range, 5-14
  start and stop columns, 5-15
partitions
  elimination, 9-35
  statistics, 8-4
partition-wise joins
  full, 5-20
  full, and EXPLAIN PLAN output, 5-20
  partial, and EXPLAIN PLAN output, 5-19
PCTFREE parameter, 2-12, 20-31
PCTINCREASE parameter, 20-37
  and SQL.BSQ file, 20-34
PCTUSED parameter, 2-12, 20-32
performance
  client/server applications, 3-11
  decision support applications, 3-4
  different types of applications, 3-2
  distributed databases, 3-8
  evaluating, 1-10
  key factors, 16-3
  mainframe, 23-6
  monitoring registered applications, 11-8
  NT, 23-6
  OLTP applications, 3-2
  Oracle Parallel Server, 3-10
  UNIX-based systems, 23-6
  viewing execution plans, 4-3
Performance Manager, 11-11
Performance Monitor
  NT, 18-5
PHYRDS column
  V$FILESTAT table, 20-17
physical reads statistic, 19-28
PHYWRTS column
  V$FILESTAT table, 20-17
ping UNIX command, 11-3
pinging, 2-13
PINS column
  V$LIBRARYCACHE table, 19-15
plan
  accessing views, 4-80, 4-82, 4-84
  complex statements, 4-75
  compound queries, 4-91, 4-92, 4-93
  joining views, 4-89
  joins, 4-49, 4-58
  OR operators, 4-73
plan stability, 10-2
  limitations of, 10-2
  preserving execution plans, 10-2
  procedures for the cost-based optimizer, 10-8
  use of hints, 10-2
PLAN_TABLE table
  BYTES column, 5-5
  CARDINALITY column, 5-5
  COST column, 5-5
  DISTRIBUTION column, 5-6
  ID column, 5-5
  OBJECT_INSTANCE column, 5-5
  OBJECT_NAME column, 5-5
  OBJECT_NODE column, 5-4
  OBJECT_OWNER column, 5-5
  OBJECT_TYPE column, 5-5
  OPERATION column, 5-4
  OPTIMIZER column, 5-5
  OPTIONS column, 5-4
  OTHER column, 5-6
  OTHER_TAG column, 5-5
  PARENT_ID column, 5-5
  PARTITION_ID column, 5-6

PARTITION_START column, 5-6
PARTITION_STOP column, 5-6
POSITION column, 5-5
REMARKS column, 5-4
SEARCH_COLUMNS column, 5-5
STATEMENT_ID column, 5-4
structure, 5-3
TIMESTAMP column, 5-4
PL/SQL
deterministic functions, 4-70
package, 11-7
tuning PL/SQL areas, 19-8
point events in Oracle Trace, 14-5
POOL attribute, 21-8
POSITION column
PLAN_TABLE table, 5-5
PQ_DISTRIBUTE hint, 7-26
PRE_PAGE_SGA parameter, 19-5
precompilers
control of parsing and private SQL areas, 19-10
predicates
optimizing view queries, 4-76
pushing into a view, 4-79, 4-84
examples, 4-80, 4-82
selectivity, 8-2
histograms, 8-17
user-defined, 4-33
PRIMARY KEY constraint, 12-10
primary keys
optimization, 4-75
searches, 4-37
private SQL areas, 19-10
PRIVATE_SGA variable, 19-24
proactive tuning, 2-2
procedures
deterministic functions, 4-70
process
dispatcher process configuration, 21-8
maximum number, 16-7
prestarting, 22-6
priority, 23-3
scheduler, 23-3
scheduling, 18-6
processing, distributed, 3-11
pseudocolumns
ROWNUM
cannot use indexes, 4-46
optimizing view queries, 4-77, 4-88
PUSH_JOIN_PRED hint, 4-87
PUSH_PRED hint, 7-34

Q

queries
avoiding the use of indexes, 12-7
compound
defined, 4-48
optimization of, 4-91
ORs converted to, 4-71
defined, 4-47
distributed, 9-30, 9-40
ensuring the use of indexes, 12-6
optimizing IN subquery, 4-78
optimizing view queries, 4-76
SAMPLE clause
cost-based optimization, 4-19
selectivity of, 4-25
star queries, 4-62
query column
SQL trace, 6-13
query plans, 5-2
query server process
tuning, 21-15

R

random reads, 20-5
random writes, 20-5
range
distribution value, 5-8
range partitions, 5-14
examples of, 5-14
raw device, 23-2
reactive tuning, 2-3
read consistency, 18-10
read/write operations, 20-5
REBUILD statement, 12-9
record keeping, 2-15
RECOVER statement
PARALLEL clause, 24-15

recovery
parallel
intra transaction recovery, 24-17
parallel processes for, 24-15
PARALLEL_MAX_SERVERS initialization parameter, 24-15
setting number of processes to use, 24-15
RECOVERY_PARALLELISM initialization parameter, 24-15
recursive calls, 6-13, 20-27
recursive SQL, 13-1
RECYCLE cache, 19-33
redo allocation latch, 21-16, 21-19
REDO BUFFER ALLOCATION RETRIES statistic, 19-7
redo copy latches, 21-16, 21-19
choosing how many, 21-16
redo logs
buffer tuning, 19-6
mirroring, 20-21
placement on disk, 20-19
reducing
buffer cache misses, 19-31
contention
dispatchers, 21-6
OS processes, 23-3
query servers, 21-15
redo log buffer latches, 21-16
shared servers, 21-9
data dictionary cache misses, 19-22
paging and swapping, 19-4
rollback segment contention, 21-4
unnecessary parse calls, 19-10
registering applications with database, 11-8
RELEASE_CURSOR clause, 19-10
RELOADS column
V$LIBRARYCACHE table, 19-15
REMARKS column
PLAN_TABLE table, 5-4
remote SQL statement, 9-30
Re-Parse Frequency data view, 14-10
resource
adding, 1-4
tuning contention, 2-13
response time, 1-2, 1-3, 4-8
cost-based approach, 4-10
optimizing, 4-9, 7-7
REWRITE hint, 7-17
RMAN
tuning for backups, 20-61
roles in tuning, 1-7
rollback segments, 18-10
assigning to transactions, 20-30
choosing how many, 21-4
contention, 21-3
creating, 21-4
detecting dynamic extension, 20-27
dynamic extension, 20-30
rollbacks
fast-start on-demand, 24-16
fast-start parallel, 24-16
round-robin
distribution value, 5-8
row sampling, 8-3
row sources, 4-6
Row Statistics for Execute/Fetch drilldown data views, 14-20
ROWID hint, 7-11
rowids
mapping to bitmaps, 12-18
table access by, 4-21
ROWNUM pseudocolumn
cannot use indexes, 4-46
optimizing view queries, 4-77, 4-88
rows
row sources, 4-6
rowids used to locate, 4-21, 4-35
rows column, SQL trace, 6-13
Rows Fetched/Fetch Count Ratio data view, 14-12
Rows Sorted data view, 14-12
RowSource event, 14-5
RULE hint, 7-8
OPTIMIZER_MODE and, 4-11
rule-based optimization, 4-34

S

SAMPLE BLOCK clause, 4-21
access path, 4-21
hints cannot override, 4-25

SAMPLE clause, 4-21
access path, 4-21
hints cannot override, 4-25
cost-based optimization, 4-19
sample table scans, 4-21
hints cannot override, 4-25
sar UNIX command, 18-5
scalability, 18-11
scans, 4-21
cluster, 4-36, 4-38
indexed, 4-39
fast full index scan, 4-23
full table, 4-21, 4-46
multiblock reads, 4-26
rule-based optimizer, 4-46
hash cluster, 4-36, 4-38
index, 4-22
bitmap, 4-24
bounded range, 4-41
cluster key, 4-39
composite, 4-39
MAX or MIN, 4-44
ORDER BY, 4-45
restrictions, 4-46
selectivity and, 4-25
single-column, 4-40
unbounded range, 4-42
index joins, 4-24
range, 4-23, 4-39, 4-40
bounded, 4-41
MAX or MIN, 4-44
ORDER BY, 4-45
unbounded, 4-42
sample table, 4-21
hints cannot override, 4-25
unique, 4-22, 4-37, 4-39
schemas
star schemas, 4-62
SEARCH_COLUMNS column
PLAN_TABLE table, 5-5
segments, 20-26
SELECT statement
SAMPLE clause, 4-21
access path, 4-21, 4-25
cost-based optimization, 4-19
selectivity of predicates, 8-2
histograms, 8-17
user-defined selectivity, 4-33
selectivity of queries, 4-25
selectivity, index, 12-4
select-project-join views, 4-76
semi-joins, 4-61
sequence cache, 2-12
sequential reads, 20-5
sequential writes, 20-5
serializable transactions, 17-6
service time, 1-2, 1-3
Session Data Unit (SDU), 22-16
session memory statistic, 19-24
SESSION_CACHED_CURSORS parameter, 19-20
SET TRANSACTION statement, 20-30
SGA size, 19-7
SGA statistics, 15-2
shared pool, 2-12
contention, 2-13
keeping objects pinned in, 13-4
tuning, 19-13, 19-25
shared SQL areas
keeping in the shared pool, 13-4
memory allocation, 19-16
similar SQL statements, 13-2
statements considered, 13-1
SHARED_POOL_RESERVED_SIZE parameter, 19-27
SHARED_POOL_SIZE parameter, 19-22, 19-27
allocating library cache, 19-16
tuning the shared pool, 19-24
short waits
definition of, 20-57
SHOW SGA statement, 19-5
Simple Network Management Protocol (SNMP), 11-5
single tier, 18-14
SNMP, 11-5
SOME, 4-67
sort areas
memory allocation, 20-35
process local area, 2-12
SORT_AREA_RETAINED_SIZE parameter, 19-42, 20-37

SORT_AREA_SIZE parameter, 4-29, 12-16, 19-41
  cost-based optimization and, 4-59
  tuning sorts, 20-36
SORT_MULTIBLOCK_READ_COUNT parameter, 20-38
sort-merge joins, 4-51
  access path, 4-43
  cost-based optimization, 4-59
  example, 4-43
sorts
  (disk) statistic, 20-35
  (memory) statistic, 20-35
  avoiding on index creation, 20-38
  tuning, 20-35
Sorts in Memory data view, 14-12
Sorts on Disk data view, 14-12
source data for tuning, 11-2
spin count, 18-12
SPINCOUNT parameter, 21-2
SQL
  functions
    optimizing view queries, 4-84
  types of statements in
    optimizing, 4-47
SQL area tuning, 19-8
SQL Parse event, 14-5
SQL statement property sheet in Oracle Trace, 14-15
SQL statements
  avoiding the use of indexes, 12-7
  complex, 4-48, 4-74
    optimizing, 4-74
  converting
    examples of, 4-71
  decomposition, 9-32
  distributed
    defined, 4-48
    optimization of, 4-94
  ensuring the use of indexes, 12-6
  execution plans of, 4-5
  matching with outlines, 10-3
  modifying indexed data, 12-5
  optimization
    complex statements, 4-74
    types of statements, 4-47
  recursive, 13-1
    OPTIMIZER_GOAL does not affect, 4-11
  remote
    defined, 4-48
  simple, 4-47
  tuning, 2-10
  types of, 4-47
SQL trace facility, 6-2, 6-6, 11-6, 19-9, 19-40
  example of output, 6-15
  output, 6-12
  parse calls, 19-9
  statement truncation, 6-14
  steps to follow, 6-3
  trace files, 6-4, 11-3
SQL*Plus script, 11-7
SQL_STATEMENT column
  TKPROF_TABLE, 6-18
SQL_TRACE parameter, 6-5
SQL.BSQ file, 20-33
STANDARD package, 13-4
STAR hint, 7-19
star joins, 4-62
star query, 4-62
star transformation, 7-34
STAR_TRANSFORMATION hint, 7-34
STAR_TRANSFORMATION_ENABLED parameter, 7-35
start columns
  in partitioning and EXPLAIN PLAN statement, 5-15
START statement in Oracle Trace, 14-21
STATEMENT_ID column
  PLAN_TABLE table, 5-4
statistics, 15-2
  consistent gets, 19-28, 21-4, 21-21
  current value, 15-4
  db block gets, 19-28, 21-4
  estimated
    block sampling, 8-3
    row sampling, 8-3
  exporting and importing, 8-2
  extensible optimization, 4-32
  from ANALYZE, 8-4
  from B*-tree or bitmap index, 8-6
  gathering with DBMS_STATS package, 8-6

generating, 8-3
generating and managing with DBMS_STATS, 8-5
generating for cost-based optimization, 8-3
HIGH_VALUE and LOW_VALUE, 4-26
max session memory, 19-24
optimizer goal, 4-11
optimizer mode, 4-10
optimizer use of, 4-10, 4-12, 8-2
partitions and subpartitions, 8-4
physical reads, 19-28
query servers, 21-15
rate of change, 15-4
selectivity of predicates, 8-2
histograms, 8-17
user-defined, 4-33
session memory, 19-24
shared server processes, 19-7, 21-9
sorts (disk), 20-35
sorts (memory), 20-35
undo block, 21-3
user-defined statistics, 4-33
STATSPACK package, 11-7, 11-8, 18-2
stop columns
in partitioning and EXPLAIN PLAN statement, 5-15
STOP statement in Oracle Trace, 14-21
storage
file, 20-5
STORAGE clause
CREATE TABLE statement, 20-24
examples, 20-24
modifying parameters, 20-33
modifying SQL.BSQ file, 20-33
OPTIMAL parameter, 20-30
stored outlines
creating and using, 10-4
execution plans and plan stability, 10-2
hints, 10-3
matching with SQL statements, 10-3
moving tables, 10-7
storage requirements, 10-4
using, 10-5
viewing data for, 10-6
stored procedures
in Oracle Trace, 14-25
registering with the database, 11-8
striping, 20-22
examples, 20-23
manual, 20-23
subpartitions
statistics, 8-4
subqueries
converting to joins, 4-74
NOT IN, 4-61
optimizing IN subquery, 4-78
subquery unnesting, 9-10
swapping, 16-5, 18-5
library cache, 19-16
reducing, 19-4
SGA, 19-42
switching processes, 18-6
symmetric multiprocessor, 9-15
System Global Area tuning, 19-4
system-specific Oracle documentation
software constraints, 16-7

T

tables
dimensions
star queries, 4-62
fact tables
star queries, 4-62
formatter in Oracle Trace, 14-5
lookup tables, 4-62
placement on disk, 20-21
striping examples, 20-23
TABLESPACE clause, 20-24
CREATE TABLE statement, 20-24
tablespaces
dictionary-mapped, 20-29
temporary, 20-37
TCP.NODELAY parameter, 22-17
TEMPORARY keyword, 20-37
temporary LOBs, 19-31
temporary tablespaces
optimizing sort, 20-37
testing, 2-14
thrashing, 18-5

thread, 23-3
throughput, 1-3, 4-8
  cost-based approach, 4-10
  optimizing, 4-9, 7-6
tiers, 18-14
TIMED_STATISTICS initialization parameter, 6-4, 23-2
  SQL Trace, 6-4
TIMESTAMP column
  PLAN_TABLE table, 5-4
TKPROF program, 6-3, 6-6, 19-40
  editing the output SQL script, 6-17
  example of output, 6-15
  generating the output SQL script, 6-16
  introduction, 11-7
  syntax, 6-8
  using the EXPLAIN PLAN statement, 6-10
TKPROF_TABLE, 6-18
  querying, 6-17
Total Elapsed Time data view, 14-11
Trace, Oracle, 14-1
transaction processing monitor, 18-15, 18-17
transactions
  assigning rollback segments, 20-30
  discrete, 17-2
  serializable, 17-6
transmission time, 16-7
Transparent Gateway, 9-40
triggers
  in tuning OLTP applications, 9-16
tuning
  access path, 2-11
  and design, 2-10
  application design, 2-9
  business rule, 2-7
  client/server applications, 3-11
  contention, 21-1
  CPU, 18-1
  data design, 2-8
  data sources, 11-2
  database logical structure, 2-9
  decision support systems, 3-4
  diagnosing problems, 16-1
  distributed databases, 3-8
  expectations, 1-9
  factors, 16-2
  goals, 1-9, 2-14
  I/O, 2-12, 20-2
  library cache, 19-14
  logical structure, 12-3
  memory allocation, 2-11, 19-2, 19-42
  method, 2-1
  monitoring registered applications, 11-8
  multi-threaded server, 21-5
  OLTP applications, 3-2
  operating system, 2-13, 16-7, 19-4
  Oracle Parallel Server, 3-10
  parallel execution, 3-5
  personnel, 1-7
  proactive, 2-2
  production systems, 2-4
  query servers, 21-15
  reactive, 2-3
  shared pool, 19-13
  sorts, 20-35
  SQL, 2-10
  SQL and PL/SQL areas, 19-8
  System Global Area (SGA), 19-4
two-tier, 18-14

U

undo block statistic, 21-3
UNION ALL operator
  examples, 4-72, 4-74, 4-91
  optimizing view queries, 4-77
  transforming OR into, 4-71
UNION ALL view, 9-36
UNION operator
  compound queries, 4-48
  examples, 4-79, 4-92
  optimizing view queries, 4-77
UNIQUE constraint, 12-10
UNIQUE index, 12-17
unique keys
  optimization, 4-75
  searches, 4-37
uniqueness, 12-10
UNIX system performance, 23-6
unlimited extents, 20-28

UNNEST hint, 7-32
UNNEST_SUBQUERY parameter, 7-32, 9-10
upgrade
  to the cost-based optimizer, 10-9
USE_CONCAT hint, 7-16
USE_MERGE hint, 7-21
USE_NL hint, 7-20
USE_STORED_OUTLINES parameter, 10-5
USER_DUMP_DEST initialization parameter, 6-4
USER_DUMP_DEST parameter
  SQL Trace parameter, 6-4
USER_ID column
  TKPROF_TABLE, 6-18
USER_INDEXES view, 12-17
USER_OULTINE_HINTS view
  stored outline hints, 10-6
USER_OUTLINES view
  stored outlines, 10-6
USER_TAB_COL_STATISTICS view, 4-26
USER_TAB_COLUMNS view, 4-26
USER_TABLES view, 4-26
user-defined costs, 4-33
users
  memory allocation, 19-8
UTLBSTAT.SQL script, 11-7
UTLCHN1.SQL script, 11-7, 20-31
UTLDTREE.SQL script, 11-7
UTLESTAT.SQL script, 11-7
UTLLOCKT.SQL script, 11-7
UTLXPLAN.SQL script, 5-3

V

V$ dynamic performance views, 11-5
V$BH view, 19-31
V$BUFFER_POOL_STATISTICS view, 19-39, 19-41
V$DATAFILE view, 20-17
V$DISPATCHER view, 21-7
V$FAST_START_SERVERS view, 24-17
V$FAST_START_TRANSACTIONS view, 24-17
V$FILESTAT view
  disk I/O, 20-17
  PHYRDS column, 20-17
  PHYWRTS column, 20-17
V$FIXED_TABLE view, 15-2
V$INSTANCE view, 15-2
V$LATCH view, 15-2, 21-2, 21-17
V$LATCH_CHILDREN view, 19-41
V$LATCH_MISSES view, 18-13
V$LIBRARYCACHE view, 15-2
  NAMESPACE column, 19-15
  PINS column, 19-15
  RELOADS column, 19-15
V$LOCK view, 15-3
V$MYSTAT view, 15-3
V$PROCESS view, 15-3
V$QUEUE view, 21-7, 21-9
V$RESOURCE_LIMIT view, 21-2
V$ROLLSTAT view, 15-2
V$ROWCACHE view, 15-2
  GETMISSES column, 19-22
  GETS column, 19-22
  performance statistics, 19-21
  using, 19-21
V$RSRC_CONSUMER_GROUP view, 18-7
V$SESSION view, 15-3
  application registration, 11-8
V$SESSION_EVENT view, 15-3
  network information, 22-9
V$SESSION_WAIT view, 15-3, 19-40, 21-2
  network information, 22-9
V$SESSTAT view, 15-3, 18-7
  network information, 22-10
  using, 19-24
V$SGA view, 15-2
V$SGASTAT view, 15-2
V$SHARED_POOL_RESERVED view, 19-27
V$SORT_USAGE view, 2-10, 15-2
V$SQL_BIND_DATA view, 19-18
V$SQL_BIND_METADATA view, 19-18
V$SQLAREA view, 15-2
  application registration, 11-8
  resource-intensive statements, 2-10
V$SQLTEXT view, 15-2
V$SYSSTAT view, 15-2, 18-6
  detecting dynamic extension, 20-27
  examining recursive calls, 20-27
  redo buffer allocation, 19-7
  tuning sorts, 20-35
  using, 19-28

V$SYSTEM_EVENT view, 15-2, 18-12, 21-2
V$WAITSTAT view, 15-2, 21-2
  reducing free list contention, 21-20
  rollback segment contention, 21-3
variables
  bind variables
    optimization, 4-27
views
  complex view merging, 4-78
  histograms, 8-21
  instance level, 15-2
  non-null values for nulls, 4-86
  optimization, 4-76
  select-project-join views, 4-76
  statistics, 8-13
  tuning, 15-1
  USER_OUTLINE_HINTS view, 10-6
  USER_OUTLINES view, 10-6
  V$FAST_START_SERVERS view, 24-17
  V$FAST_START_TRANSACTIONS view, 24-17
vmstat UNIX command, 18-5

W

wait detection, 18-11
wait time, 1-3, 1-4
Waits by Average Wait Time data view, 14-13
Waits by Event Frequency data view, 14-13
Waits by Total Wait Time data view, 14-13
workload, 1-6
write batch size, 20-45