Oracle9*i*

Database Performance Guide and Reference

Release 1 (9.0.1)

June 2001

Part No. A87503-02

ORACLE®

Oracle9i Database Performance Guide and Reference, Release 1 (9.0.1)

Part No. A87503-02

Primary Author: Michele Cyran

Contributing Author: Connie Dialeris Green

Contributors: Rafi Ahmed, Ahmed Alomari, Hermann Baer, Ruth Baylis, Leo Cloutier, Maria Colgan, Dinish Das, Lex de Haan, Harv Eneman, Bjorn Engsig, Cecilia Gervasio, Leslie Gloyd, Karl Haas, Andrew Holdsworth, Mamdouh Ibrahim, Namit Jain, Hakan Jakobsson, Sanjay Kaluskar, Srinivas Kareenhalli, Peter Kilpatrick, Sushil Kumar, Tirthankar Lahiri, Yunrui Li, Diana Lorentz, Roderick Manalac, Alan Maxwell, Joe McDonald, Ari Mozes, Gary Ngai, Arvind Nithrakashyap, Peter Povinec, Richard Powell, Shankar Raman, Virag Saksena, Slartibartfast, Vinay Srihari, Frank Stephens, Mike Stewart, Mineharu Takahara, Patrick Tearle, Nitin Vengurlekar, Jean Francois Verrier, Steve Vivian, Simon Watt, Sabrina Whitehouse, Graham Wood, Mohamed Ziauddin

Contents

Send Us Your Comments xix

Preface xxi

Audience xxii
Organization xxii
Related Documentation xxvi
Conventions xxvii
Documentation Accessibility xxix

What's New in Oracle Performance? xxxi

Part I Writing and Tuning SQL

1 Introduction to the Optimizer

Overview of SQL Processing Architecture 1-2
Overview of the Optimizer 1-3
Steps in Optimizer Operations 1-4
Understanding Execution Plans 1-5
Choosing an Optimizer Approach and Goal 1-10
How the CBO Optimizes SQL Statements for Fast Response 1-13
Features that Require the CBO 1-14
Understanding the Cost-Based Optimizer (CBO) 1-15
Architecture of the CBO 1-16
Understanding Access Paths for the CBO 1-23

Full Table Scans ... 1-24
Sample Table Scans ... 1-27
ROWID Scans ... 1-28
Index Scans ... 1-29
Cluster Scans ... 1-39
Hash Scans ... 1-39
How the CBO Chooses an Access Path ... 1-39
Understanding Joins ... 1-42
How the CBO Executes Join Statements ... 1-42
How the CBO Chooses the Join Method ... 1-43
Nested Loop Joins ... 1-44
Nested Loop Outer Joins ... 1-48
Hash Joins ... 1-49
Hash Join Outer Joins ... 1-51
Sort Merge Joins ... 1-53
Sort Merge Outer Joins ... 1-57
Cartesian Joins ... 1-57
Full Outer Joins ... 1-62
How the CBO Chooses Execution Plans for Joins ... 1-63
How the CBO Executes Anti-Joins ... 1-64
How the CBO Executes Semi-Joins ... 1-64
How the CBO Executes Star Queries ... 1-65
Cost-Based Optimizer Parameters ... 1-66
OPTIMIZER_FEATURES_ENABLE Parameter ... 1-66
Other CBO Parameters ... 1-68
Overview of the Extensible Optimizer ... 1-72
Understanding User-Defined Statistics ... 1-72
Understanding User-Defined Selectivity ... 1-73
Understanding User-Defined Costs ... 1-73

2 Optimizer Operations

How the Optimizer Performs Operations ... 2-2
How the CBO Evaluates IN-List Iterators ... 2-2
How the CBO Evaluates Concatenation ... 2-6
How the CBO Evaluates Remote Operations ... 2-9

How the CBO Executes Distributed Statements 2-12
How the CBO Executes Sort Operations 2-13
How the CBO Executes Views 2-16
How the CBO Evaluates Constants 2-18
How the CBO Evaluates the UNION / UNION ALL Operators 2-19
How the CBO Evaluates the LIKE Operator 2-20
How the CBO Evaluates the IN Operator 2-21
How the CBO Evaluates the ANY or SOME Operator 2-21
How the CBO Evaluates the ALL Operator 2-22
How the CBO Evaluates the BETWEEN Operator 2-22
How the CBO Evaluates the NOT Operator 2-23
How the CBO Evaluates Transitivity 2-23
How the CBO Optimizes Common Subexpressions 2-24
How the CBO Evaluates DETERMINISTIC Functions 2-26
How the Optimizer Transforms SQL Statements 2-27
How the CBO Transforms ORs into Compound Queries 2-28
How the CBO Unnests Subqueries 2-31
How the CBO Merges Views 2-33
How the CBO Pushes Predicates 2-36
How the CBO Executes Compound Queries 2-46

3 Gathering Optimizer Statistics

Understanding Statistics 3-2
Generating Statistics 3-3
Using the DBMS_STATS Package 3-5
Using the ANALYZE Statement 3-12
Finding Data Distribution 3-13
Missing Statistics 3-13
Using Statistics 3-14
Managing Statistics 3-14
Verifying Table Statistics 3-16
Verifying Index Statistics 3-17
Verifying Column Statistics 3-18
Using Histograms 3-20
When to Use Histograms 3-21

Creating Histograms 3-21
Types of Histograms 3-22
Viewing Histograms 3-24
Verifying Histogram Statistics 3-24

4 Understanding Indexes and Clusters

Understanding Indexes 4-2
Tuning the Logical Structure 4-2
Choosing Columns and Expressions to Index 4-3
Choosing Composite Indexes 4-4
Writing Statements that Use Indexes 4-6
Writing Statements that Avoid Using Indexes 4-6
Re-creating Indexes 4-6
Compacting Indexes 4-7
Using Nonunique Indexes to Enforce Uniqueness 4-8
Using Enabled Novalidated Constraints 4-8
Using Function-based Indexes 4-9
Using Index-Organized Tables 4-11
Using Bitmap Indexes 4-11
When to Use Bitmap Indexes 4-12
Using Bitmap Indexes with Good Performance 4-14
Initialization Parameters for Bitmap Indexing 4-16
Using Bitmap Access Plans on Regular B-tree Indexes 4-17
Bitmap Index Restrictions 4-18
Using Bitmap Join Indexes 4-18
Using Domain Indexes 4-18
Using Clusters 4-19
Using Hash Clusters 4-20

5 Optimizer Hints

Understanding Optimizer Hints 5-2
Specifying Hints 5-2
Using Optimizer Hints 5-6
Hints for Optimization Approaches and Goals 5-6
Hints for Access Paths 5-9

Hints for Query Transformations 5-17
Hints for Join Orders 5-21
Hints for Join Operations 5-23
Hints for Parallel Execution 5-28
Additional Hints 5-33
Using Hints with Views 5-38

6 Optimizing SQL Statements

Goals for Tuning 6-2
Reduce the Workload 6-2
Balance the Workload 6-2
Parallelize the Workload 6-2
Identifying and Gathering Data on Resource-Intensive SQL 6-3
Identifying Resource-Intensive SQL 6-3
Gathering Data on the SQL Identified 6-4
Overview of SQL Statement Tuning 6-5
Verifying Optimizer Statistics 6-6
Reviewing the Execution Plan 6-7
Restructuring the SQL Statements 6-7
Controlling the Access Path and Join Order with Hints 6-16
Restructuring the Indexes 6-20
Modifying or Disabling Triggers and Constraints 6-20
Restructuring the Data 6-20
Maintaining Execution Plans Over Time 6-21
Visiting Data as Few Times as Possible 6-21

7 Using Plan Stability

Using Plan Stability to Preserve Execution Plans 7-2
Using Hints with Plan Stability 7-2
Storing Outlines 7-4
Enabling Plan Stability 7-4
Using Supplied Packages to Manage Stored Outlines 7-4
Creating Outlines 7-4
Using and Editing Stored Outlines 7-6
Viewing Outline Data 7-9

Moving Outline Tables 7-9
Using Plan Stability with the Cost-Based Optimizer 7-11
Using Outlines to Move to the Cost-Based Optimizer 7-11
Upgrading and the Cost-Based Optimizer 7-12

8 Using the Rule-Based Optimizer

Overview of the Rule-Based Optimizer (RBO) 8-2
Understanding Access Paths for the RBO 8-2
Choosing Execution Plans for Joins with the RBO 8-15
Transforming and Optimizing Statements with the RBO 8-17
Transforming ORs into Compound Queries with the RBO 8-17
Using Alternative SQL Syntax 8-20

Part II SQL-Related Performance Tools

9 Using EXPLAIN PLAN

Understanding EXPLAIN PLAN 9-2
How Execution Plans Can Change 9-2
Looking Beyond Execution Plans 9-3
Creating the PLAN_TABLE Output Table 9-4
Running EXPLAIN PLAN 9-4
Identifying Statements for EXPLAIN PLAN 9-4
Specifying Different Tables for EXPLAIN PLAN 9-5
Displaying PLAN_TABLE Output 9-5
Reading EXPLAIN PLAN Output 9-6
Viewing Bitmap Indexes with EXPLAIN PLAN 9-10
Viewing Partitioned Objects with EXPLAIN PLAN 9-11
Examples of Displaying Range and Hash Partitioning with EXPLAIN PLAN 9-11
Examples of Pruning Information with Composite Partitioned Objects 9-13
Examples of Partial Partition-wise Joins 9-16
Examples of Full Partition-wise Joins 9-17
Examples of INLIST ITERATOR and EXPLAIN PLAN 9-18
Example of Domain Indexes and EXPLAIN PLAN 9-19
EXPLAIN PLAN Restrictions 9-20

PLAN_TABLE Columns 9-20

10 Using SQL Trace and TKPROF

Understanding SQL Trace and TKPROF 10-2
Understanding the SQL Trace Facility 10-2
Understanding TKPROF 10-3
Using the SQL Trace Facility and TKPROF 10-3
Step 1: Setting Initialization Parameters for Trace File Management 10-4
Step 2: Enabling the SQL Trace Facility 10-5
Step 3: Formatting Trace Files with TKPROF 10-6
Step 4: Interpreting TKPROF Output 10-11
Step 5: Storing SQL Trace Facility Statistics 10-16
Avoiding Pitfalls in TKPROF Interpretation 10-19
Avoiding the Argument Trap 10-19
Avoiding the Read Consistency Trap 10-19
Avoiding the Schema Trap 10-20
Avoiding the Time Trap 10-21
Avoiding the Trigger Trap 10-22
Sample TKPROF Output 10-22
Sample TKPROF Header 10-23
Sample TKPROF Body 10-23
Sample TKPROF Summary 10-29

11 Using Autotrace in SQL*Plus

Controlling the Autotrace Report 11-2
Execution Plan 11-2
Statistics 11-3
Tracing Parallel and Distributed Queries 11-6
Monitoring Disk Reads and Buffer Gets 11-8

12 Using Oracle Trace

Overview of Oracle Trace 12-2
Event Data 12-2
Event Sets 12-2

Accessing Collected Data 12-3
Collecting Oracle Trace Data 12-3
Using the Oracle Trace Command-Line Interface 12-3
Using Initialization Parameters to Control Oracle Trace 12-7
Controlling Oracle Trace Collections from PL/SQL 12-10
Accessing Oracle Trace Collection Results 12-12
Formatting Oracle Trace Data to Oracle Tables 12-13
Running the Oracle Trace Reporting Utility 12-14
Oracle Server Events 12-15
Data Items Collected for Events 12-16
Items Associated with Each Event 12-22
Troubleshooting Oracle Trace 12-32
Oracle Trace Configuration 12-32
Formatter Tables 12-36

Part III Creating a Database for Good Performance

13 Building a Database for Performance

Initial Database Creation 13-2
Database Creation using the Installer 13-2
Manual Database Creation 13-2
Parameters Necessary for Initial Database Creation 13-2
The CREATE DATABASE Statement 13-3
Running Data Dictionary Scripts 13-4
Sizing Redo Log Files 13-5
Creating Subsequent Tablespaces 13-6
Creating Tables for Good Performance 13-6
Loading and Indexing Data 13-8
Using SQL*Loader for Good Performance 13-8
Efficient Index Creation 13-8
Initial Instance Configuration 13-9
Configuring Rollback Segments 13-11
Setting up OS, Database, and Network Monitoring 13-12

14 Memory Configuration and Use

Understanding Memory Allocation Issues 14-2
Oracle Memory Caches 14-2
Dynamically Changing Cache Sizes 14-2
Application Considerations 14-3
Operating System Memory Use 14-3
Iteration During Configuration 14-4
Configuring and Using the Buffer Cache 14-5
Using the Buffer Cache Effectively 14-5
Sizing the Buffer Cache 14-5
Interpreting and Using the Buffer Cache Statistics 14-9
Considering Multiple Buffer Pools 14-11
Buffer Pool data in V$DB_CACHE_ADVICE 14-13
Buffer Pool Hit Ratios 14-13
Determining Which Segments Have Many Buffers in the Pool 14-13
Keep Pool 14-15
Recycle Pool 14-16
Configuring and Using the Shared Pool and Large Pool 14-17
Shared Pool Concepts 14-18
Using the Shared Pool Effectively 14-22
Sizing the Shared Pool 14-26
Interpreting Shared Pool Statistics 14-32
Consider using the Large Pool 14-33
Consider Using CURSOR_SPACE_FOR_TIME 14-37
Consider Caching Session Cursors 14-38
Consider Configuring the Reserved Pool 14-38
Consider Keeping Large Objects to Prevent Aging 14-41
Consider CURSOR_SHARING for Existing Applications 14-42
Configuring and Using the Java Pool 14-44
Configuring and Using the Redo Log Buffer 14-44
Sizing the Log Buffer 14-45
Log Buffer Statistics 14-45
Configuring the PGA Working Memory 14-46
Automatic PGA Memory Management 14-48
Configuring SORT_AREA_SIZE 14-54

Reducing Total Memory Usage 14-60

15 I/O Configuration and Design

Understanding I/O 15-2
Basic I/O Configuration 15-5
Determining Application I/O Characteristics 15-6
I/O Configuration Decisions 15-9
Know your I/O System 15-9
Match I/O Requirements with the I/O System 15-10
Layout the Files using Operating System or Hardware Striping 15-11
Manually Distributing I/O 15-14
When to Separate Files 15-15
Three Sample Configurations 15-17
Oracle-Managed Files 15-18
Choosing Data Block Size 15-19

16 Understanding Operating System Resources

Understanding Operating System Performance Issues 16-2
Using Hardware and Operating System Caches 16-2
Evaluating Raw Devices 16-2
Using Process Schedulers 16-3
Using Operating System Resource Managers 16-3
Solving Operating System Problems 16-5
Performance Hints on UNIX-Based Systems 16-5
Performance Hints on NT Systems 16-6
Performance Hints on Mainframe Computers 16-6
Understanding CPU 16-7
Finding System CPU Utilization 16-9

17 Configuring Instance Recovery Performance

Understanding Instance Recovery 17-2
Checkpointing and Cache Recovery 17-2
How Checkpoints Affect Performance 17-3
Reducing Checkpoint Frequency to Optimize Runtime Performance 17-3

Configuring the Duration of Cache Recovery 17-4
Use Fast-Start Checkpointing to Limit Instance Recovery Time 17-4
Set LOG_CHECKPOINT_TIMEOUT to Influence the Number of Redo Logs 17-6
Set LOG_CHECKPOINT_INTERVAL to Influence the Number of Redo Logs 17-6
Use Parallel Recovery to Speed up Redo Application 17-7
Initialization Parameters that Influence Cache Recovery Time 17-7
Monitoring Cache Recovery 17-8
Tuning Transaction Recovery 17-15

18 Configuring Undo and Temporary Segments

Configuring Undo Segments 18-2
Configuring Automatic Undo Management 18-2
Configuring Rollback Segments 18-2
Configuring Temporary Tablespaces 18-4

19 Configuring Shared Servers

Configuring the Number of Shared Servers 19-2
Identifying Contention Using the Dispatcher-Specific Views 19-2
Reducing Contention for Dispatcher Processes 19-3
Reducing Contention for Shared Servers 19-5
Determining the Optimal Number of Dispatchers and Shared Servers 19-9

Part IV System-Related Performance Tools

20 Oracle Tools to Gather Database Statistics

Overview of Tools 20-2
Principles of Data Gathering 20-2
Interpreting Statistics 20-3
Oracle Enterprise Manager Diagnostics Pack 20-4
Statspack 20-5
V$ Performance Views 20-6

21 Using Statspack

Statspack vs. BSTAT/ESTAT 21-2

How Statspack Works 21-3
Configuring Statspack 21-3
Database Space Requirements for Statspack 21-3
Statspack in Dictionary Managed Tablespaces 21-4
Statspack in Locally Managed Tablespaces 21-4
Installing Statspack 21-4
Interactive Statspack Installation 21-4
Batch Mode Statspack Installation 21-6
Using Statspack 21-6
Taking a Statspack Snapshot 21-6
Running a Statspack Performance Report 21-8
Configuring the Amount of Data Captured in Statspack 21-14
Time Units Used for Wait Events 21-18
Event Timings 21-19
Managing and Sharing Statspack Performance Data 21-19
Oracle Real Application Clusters Considerations with Statspack 21-22
Removing Statspack 21-23
Statspack Supplied Scripts 21-24
Scripts for Statspack Installation 21-24
Scripts for Statspack Reporting and Automation 21-24
Scripts for Upgrading Statspack 21-24
Scripts for Statspack Performance Data Maintenance 21-25
Scripts for Statspack Documentation 21-25
Statspack Limitations 21-25

Part V Optimizing Instance Performance

22 Instance Tuning

Performance Tuning Principles 22-2
Baselines 22-2
The Symptoms and the Problems 22-3
When to Tune 22-3
Performance Tuning Steps 22-4
Define the Problem 22-5
Examine the Host System 22-6

Examine the Oracle Statistics 22-9
Implement and Measure Change 22-11
Interpreting Oracle Statistics 22-12
Examine Load 22-12
Using Wait Event Statistics to Drill Down to Bottlenecks 22-13
Table of Wait Events and Potential Causes 22-15
Additional Statistics 22-16
Wait Events 22-19
SQL*Net 22-20
buffer busy waits 22-22
db file scattered read 22-25
db file sequential read 22-27
direct path read 22-28
direct path write 22-30
enqueue 22-31
free buffer waits 22-34
latch free 22-36
log buffer space 22-41
log file switch 22-41
log file sync 22-42
rdbms ipc reply 22-43
Idle Wait Events 22-43

23 Tuning Networks

Understanding Connection Models 23-2
Detecting Network Problems 23-5
Using Dynamic Performance Views for Network Performance 23-6
Understanding Latency and Bandwidth 23-6
Solving Network Problems 23-8
Finding Network Bottlenecks 23-8
Dissecting Network Bottlenecks 23-10
Using Array Interfaces 23-13
Adjusting Session Data Unit Buffer Size 23-14
Using TCP.NODELAY 23-14
Using Connection Manager 23-14

Part VI Performance-Related Reference Information

24 Dynamic Performance Views for Tuning

Current State Views 24-2
Counter/Accumulator Views 24-2
Information Views 24-3
V$DB_OBJECT_CACHE 24-4
V$FILESTAT 24-6
V$LATCH 24-9
V$LATCH_CHILDREN 24-12
V$LATCH_HOLDER 24-13
V$LIBRARYCACHE 24-15
V$LOCK 24-17
V$MYSTAT 24-22
V$OPEN_CURSOR 24-23
V$PARAMETER and V$SYSTEM_PARAMETER 24-25
V$PROCESS 24-27
V$ROLLSTAT 24-29
V$ROWCACHE 24-30
V$SESSION 24-32
V$SESSION_EVENT 24-36
V$SESSION_WAIT 24-38
V$SESSTAT 24-42
V$SQL 24-46
V$SQL_PLAN 24-47
V$SQLAREA 24-53
V$SQLTEXT 24-56
V$SYSSTAT 24-57
V$SYSTEM_EVENT 24-63
V$UNDOSTAT 24-66
V$WAITSTAT 24-68

A Schemas Used in Performance Examples

Glossary

Index

Send Us Your Comments

Oracle9i Database Performance Guide and Reference, Release 1 (9.0.1)

Part No. A87503-02

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

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

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

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

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

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

Preface

This book describes detailed ways to enhance Oracle performance by writing and tuning SQL properly, using performance tools, and optimizing instance performance. It also explains how to create an initial database for good performance and includes performance-related reference information.

This preface contains these topics:

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

Audience

Oracle9i Database Performance Guide and Reference is an aid for people responsible for the operation, maintenance, and performance of Oracle. This could be useful for database administrators, application designers, and programmers. Readers should be familiar with Oracle9*i*, *Oracle9i Database Performance Methods*, the operating system, and application design before reading this manual.

Many client / server application programmers consider SQL a messaging language, because queries are issued and data is returned. However, client tools often generate inefficient SQL statements. Therefore, a good understanding of the database SQL processing engine is necessary for writing optimal SQL. This is especially true for high transaction processing systems.

Typically, SQL statements issued by OLTP applications operate on relatively few rows at a time. If an index can point to the exact rows that you want, then Oracle can construct an accurate plan to access those rows efficiently via the shortest possible path.

In DSS environments, selectivity is less important, because they often access most of a table's rows. In such situations, full table scans are common, and indexes are not even used.

This book is primarily focussed on OLTP-type applications. For detailed information on DSS and mixed environments, see the *Oracle9i Data Warehousing Guide*.

Organization

This document contains:

Part I, "Writing and Tuning SQL"

This section provides information to help understand and manage SQL statements.

Chapter 1, "Introduction to the Optimizer"

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

Chapter 2, "Optimizer Operations"

This chapter provides details of how the CBO provides specific operations.

Chapter 3, "Gathering Optimizer Statistics"

This chapter explains why statistics are important for the cost-based optimizer and describes how to gather and use statistics.

Chapter 4, "Understanding Indexes and Clusters"

This chapter describes how to create indexes and clusters, and when to use them.

Chapter 5, "Optimizer Hints"

This chapter offers recommendations on how to use cost-based optimizer hints to enhance Oracle performance.

Chapter 6, "Optimizing SQL Statements"

This chapter describes how Oracle optimizes SQL using the cost-based optimizer (CBO).

Chapter 7, "Using Plan Stability"

This chapter describes how to use plan stability (stored outlines) to preserve performance characteristics.

Chapter 8, "Using the Rule-Based Optimizer"

This chapter discusses Oracle's rule-based optimizer (RBO).

Part II, "SQL-Related Performance Tools"

This section provides information about Oracle SQL-related performance tools.

Chapter 9, "Using EXPLAIN PLAN"

This chapter shows how to use the SQL statement `EXPLAIN PLAN` and format its output.

Chapter 10, "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 11, "Using Autotrace in SQL*Plus"

This chapter describes the use of Autotrace, which can automatically get reports on the execution path used by the SQL optimizer and the statement execution statistics to help you monitor and tune statement performance.

Chapter 12, "Using Oracle Trace"

This chapter provides an overview of Oracle Trace usage and describes the Oracle Trace initialization parameters.

Part III, "Creating a Database for Good Performance"

This section describes how to create and configure a database for good performance.

Chapter 13, "Building a Database for Performance"

This chapter describes how to design and create a database for the intended needs.

Chapter 14, "Memory Configuration and Use"

This chapter explains how to allocate memory to database structures.

Chapter 15, "I/O Configuration and Design"

This chapter introduces fundamental I/O concepts, discusses the I/O requirements of different parts of the database, and provides sample configurations for I/O subsystem design.

Chapter 16, "Understanding Operating System Resources"

This chapter explains how to tune the operating system for optimal performance of Oracle.

Chapter 17, "Configuring Instance Recovery Performance"

This chapter explains how to tune recovery performance.

Chapter 18, "Configuring Undo and Temporary Segments"

This chapter explains how to configure undo segments (using automatic undo management or using rollback segments) and how to configure temporary tablespaces.

Chapter 19, "Configuring Shared Servers"

This chapter explains how to identify and reduce contention for dispatcher processes and for shared servers.

Part IV, "System-Related Performance Tools"

This section provides information about Oracle's system-related performance tools.

Chapter 20, "Oracle Tools to Gather Database Statistics"

Oracle provides a number of tools that allow a performance engineer to gather information regarding instance and database performance. This chapter explains why performance data gathering is important, and it describes how to use available tools.

Chapter 21, "Using Statspack"

This chapter describes the use of Statspack to collect, store, and analyze system data.

Part V, "Optimizing Instance Performance"

This section describes how to tune various elements of a database system to optimize performance of an Oracle instance.

Chapter 22, "Instance Tuning"

This chapter discusses the method used for performing tuning. It also describes Oracle statistics and wait events.

Chapter 23, "Tuning Networks"

This chapter describes different connection models and networking issues that affect tuning.

Part VI, "Performance-Related Reference Information"

This section provides reference information regarding dynamic performance views and wait events.

Chapter 24, "Dynamic Performance Views for Tuning"

This chapter provides detailed information on several dynamic performance views that can help you tune your system and investigate performance problems.

Appendix

Appendix A, "Schemas Used in Performance Examples"

This appendix describes the tables used in examples in Chapter 1, "Introduction to the Optimizer" and Chapter 9, "Using EXPLAIN PLAN".

Related Documentation

Before reading this manual, you should have already read *Oracle9i Database Performance Methods*, *Oracle9i Database Concepts*, the *Oracle9i Application Developer's Guide - Fundamentals*, and the *Oracle9i Database 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*, *Oracle Enterprise Manager Performance Monitoring and Planning Guide*, and the *Database Tuning with the Oracle Tuning Pack.*

For more information about tuning data warehouse environments, see the *Oracle9i Data Warehousing Guide.*

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

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

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

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

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

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

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

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

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

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

Conventions

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

- Conventions in Text
- Conventions in Code Examples

Conventions in Text

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

Convention	Meaning	Example
Bold	Bold typeface indicates terms that are defined in the text or terms that appear in a glossary, or both.	When you specify this clause, you create an **index-organized table**.
Italics	Italic typeface indicates book titles, emphasis, syntax clauses, or placeholders.	*Oracle9i Database Concepts* You can specify the *parallel_clause*. Run U*old_release*.SQL where *old_release* refers to the release you installed prior to upgrading.
UPPERCASE monospace (fixed-width font)	Uppercase monospace typeface indicates elements supplied by the system. Such elements include parameters, privileges, datatypes, RMAN keywords, SQL keywords, SQL*Plus or utility commands, packages and methods, as well as system-supplied column names, database objects and structures, user names, and roles.	You can specify this clause only for a NUMBER column. You can back up the database using the BACKUP command. Query the TABLE_NAME column in the USER_TABLES data dictionary view. Specify the ROLLBACK_SEGMENTS parameter. Use the DBMS_STATS.GENERATE_STATS procedure.
lowercase monospace (fixed-width font)	Lowercase monospace typeface indicates executables and sample user-supplied elements. Such elements include computer and database names, net service names, and connect identifiers, as well as user-supplied database objects and structures, column names, packages and classes, user names and roles, program units, and parameter values.	Enter sqlplus to open SQL*Plus. The department_id, department_name, and location_id columns are in the hr.departments table. Set the QUERY_REWRITE_ENABLED initialization parameter to true. Connect as oe user.

Conventions in Code Examples

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

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

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

Convention	Meaning	Example
[ ]	Brackets enclose one or more optional items. Do not enter the brackets.	`DECIMAL (`*`digits`*` [ , `*`precision`*` ])`
{ }	Braces enclose two or more items, one of which is required. Do not enter the braces.	`{ENABLE | DISABLE}`
|	A vertical bar represents a choice of two or more options within brackets or braces. Enter one of the options. Do not enter the vertical bar.	`{ENABLE | DISABLE}` `[COMPRESS | NOCOMPRESS]`
...	Horizontal ellipsis points indicate either:	
	■ That we have omitted parts of the code that are not directly related to the example	`CREATE TABLE ... AS `*`subquery`*`;`
	■ That you can repeat a portion of the code	`SELECT `*`col1`*`, `*`col2`*`, ... , `*`coln`*` FROM employees;`
. . .	Vertical ellipsis points indicate that we have omitted several lines of code not directly related to the example.	
Other notation	You must enter symbols other than brackets, braces, vertical bars, and ellipsis points as shown.	`acctbal NUMBER(11,2);` `acct CONSTANT NUMBER(4) := 3;`
Italics	Italicized text indicates placeholders or variables for which you must supply particular values.	`CONNECT SYSTEM/`*`system_password`* `DB_NAME = `*`database_name`*

Convention	Meaning	Example

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

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

Documentation Accessibility

Oracle's goal is to make our products, services, and supporting documentation accessible to the disabled community with good usability. To that end, our documentation includes features that make information available to users of assistive technology. This documentation is available in HTML format, and contains markup to facilitate access by the disabled community. Standards will continue to evolve over time, and Oracle is actively engaged with other market-leading technology vendors to address technical obstacles so that our documentation can be accessible to all of our customers. For additional information, visit the Oracle Accessibility Program Web site at

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

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

What's New in Oracle Performance?

This section describes new performance features of Oracle9*i* release and provides pointers to additional information. The features and enhancements described in this section comprise the overall effort to optimize server performance.

> **Note:** The new title of this book reflects the book's new design as a performance reference.

- **FIRST_ROWS_*n* Optimization**

  You can now set the `OPTIMIZER_MODE` initialization parameter to this value. With this, the optimizer uses a cost-based approach, regardless of the presence of statistics, and optimizes with a goal of best response time to return first *n* number of rows (where *n* can equal 1, 10, 100, or 1000). There is also a new `FIRST_ROWS(`*n*`)` hint (where *n* can be any positive integer). `FIRST_ROWS` is available for backward compatibility and plan stability.

  See Also: Chapter 1, "Introduction to the Optimizer"

- **Cursor Sharing Enhancements**

  The CBO now peeks at the values of user-defined bind variables on the first invocation of a cursor. This lets the optimizer determine the selectivity of any `WHERE` clause condition, as well as if literals had been used instead of bind variables. When bind variables are used in a statement, it is assumed that cursor sharing is intended and that different invocations are supposed to use the same execution plan.

  Also, the `CURSOR_SHARING` parameter can now be set to `SIMILAR` to force 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 difference between SIMILAR and FORCE is that SIMILAR forces similar statements to share the SQL area without deteriorating execution plans. There is also a new hint: CURSOR_SHARING_EXACT.

> **See Also:** Chapter 1, "Introduction to the Optimizer", Chapter 14, "Memory Configuration and Use", and Chapter 5, "Optimizer Hints"

- **Identifying Unused Indexes**

  You can find indexes that are not being used by using the ALTER INDEX MONITORING USAGE functionality over a period of time that is representative of your workload.

  > **See Also:** Chapter 4, "Understanding Indexes and Clusters"

- **System Statistics**

  You can now gather system statistics, which allow the optimizer to consider a system's I/O and CPU performance and utilization. For each plan candidate, the optimizer computes estimates for I/O and CPU costs. It is important to know the system characteristics to pick the most efficient plan with optimal proportion between I/O and CPU cost.

  > **See Also:** Chapter 3, "Gathering Optimizer Statistics"

- **Optimizer Hints**

  The following hints are new with 9*i*: NL_AJ, NL_SJ, CURSOR_SHARING_EXACT, FACT, NO_FACT and FIRST_ROWS(*n*).

  > **See Also:** Chapter 5, "Optimizer Hints"

- **Outline Editing**

  You can now edit private outlines. While the optimizer usually chooses optimal plans for queries, there are times when users know things about the execution environment that are inconsistent with the heuristics that the optimizer follows. By editing outlines directly, you can tune the SQL query without having to alter the application.

The DBMS_OUTLN package (synonym for OUTLN_PKG) and the new DBMS_OUTLN_EDIT package provides procedures used for managing stored outlines and their outline categories.

See Also: Chapter 7, "Using Plan Stability"

- **CPU Costing**

  The optimizer now calculates the cost of access paths and join orders based on the estimated computer resources, including I/O, CPU, and memory.

  See Also: Chapter 1, "Introduction to the Optimizer" and Chapter 9, "Using EXPLAIN PLAN"

- **Building a Database for Performance**

  This new chapter describes how to design and create a database for the intended needs.

  See Also: Chapter 13, "Building a Database for Performance"

- **Tuning Oracle-Managed Files**

  For systems where a file system can be used to contain all Oracle data, database administration is simplified by using Oracle-managed files. Oracle internally uses standard file system interfaces to create and delete files as needed for tablespaces, tempfiles, online logs, and controlfiles.

  See Also: Chapter 15, "I/O Configuration and Design"

- **FAST_START_MTTR_TARGET Parameter**

  You can now specify in seconds the expected "mean time to recover" (MTTR), which is the expected amount of time Oracle takes to perform recovery for an instance. The FAST_START_IO_TARGET parameter has been deprecated and should not be used.

  See Also: Chapter 17, "Configuring Instance Recovery Performance"

- **Statspack**

  The Statspack package builds off the traditional UTLBSTAT/UTLESTAT tuning scripts. Statspack automates the gathering of data, stores data and statistics, and

generates performance reports. Statspack takes "snapshots" of data to work with, letting you choose the snapshot levels and thresholds to be used.

See Also: Chapter 21, "Using Statspack"

- **Oracle Trace**

  There are new sections describing Oracle Trace events, event sets, data items specific to the Oracle9*i* server, and Oracle Trace troubleshooting. The section on methods available for collecting Oracle Trace data has been reorganized. Oracle Trace Manager is no longer shipped, and the Data Viewer GUI tool is now described in Enterprise Manager documents.

  See Also: Chapter 12, "Using Oracle Trace"

- **SQL Working Memory Management**

  With release 9*i*, it is now possible to simplify and improve the way the PGA is allocated in DSS systems. There is an automatic mode to adjust the size of the tunable portion of the PGA memory allocated by an instance dynamically. The size of that tunable portion is adjusted based on an overall PGA memory target explicitly set by the DBA.

  See Also: Chapter 14, "Memory Configuration and Use"

- **Dynamic SGA**

  It is now possible to resize the buffer cache and shared pool dynamically, while the instance is running.

  See Also: Chapter 14, "Memory Configuration and Use"

- **Dynamic Performance Views Reference**

  This chapter is new in this book. It provides detailed information on the most important dynamic performance views that can help you tune your system and investigate performance problems.

  See Also: Chapter 24, "Dynamic Performance Views for Tuning"

Part I

Writing and Tuning SQL

Part I provides information on understanding and managing your SQL statements for best performance. It is best to read these chapters in the order in which they are presented.

The chapters in this part are:

- Chapter 1, "Introduction to the Optimizer"
- Chapter 2, "Optimizer Operations"
- Chapter 3, "Gathering Optimizer Statistics"
- Chapter 4, "Understanding Indexes and Clusters"
- Chapter 5, "Optimizer Hints"
- Chapter 6, "Optimizing SQL Statements"
- Chapter 7, "Using Plan Stability"
- Chapter 8, "Using the Rule-Based Optimizer"

1

Introduction to the Optimizer

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

This chapter contains the following sections:

- Overview of SQL Processing Architecture
- Overview of the Optimizer
- Choosing an Optimizer Approach and Goal
- Understanding the Cost-Based Optimizer (CBO)
- Understanding Access Paths for the CBO
- Understanding Joins
- Cost-Based Optimizer Parameters
- Overview of the Extensible Optimizer

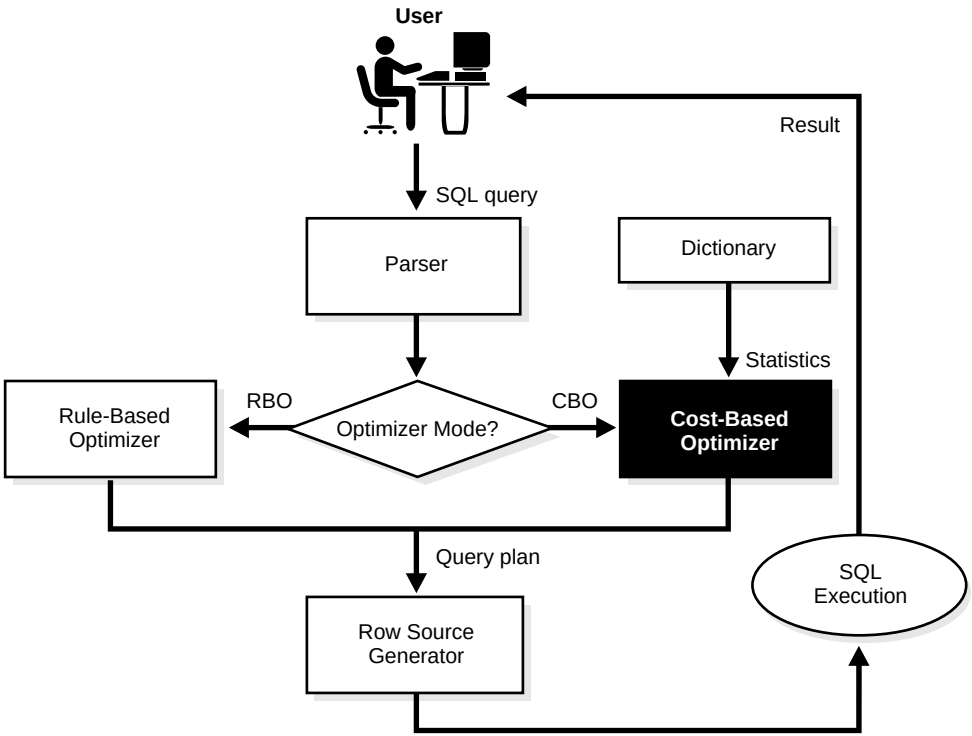
Overview of SQL Processing Architecture

The SQL processing architecture contains the following main components:

- Parser
- Optimizer
- Row Source Generator
- SQL Execution Engine

Figure 1–1 illustrates the SQL processing architecture:

Figure 1–1 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 uses internal rules or costing methods to determine the most efficient way of producing the result of the query. The output from the optimizer is a plan that describes an optimum method of execution. The Oracle server provides two methods of optimization: cost-based optimizer (CBO) and rule-based optimizer (RBO).

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. Each row source returns a set of rows for that step.

SQL Execution Engine

SQL execution is the component that operates on the execution plan associated with a SQL statement. It then produces the results of the query. Each row source produced by the row source generator is executed by the SQL execution engine.

Overview of the Optimizer

The optimizer determines the most efficient way to execute a SQL statement. This is an important step in the processing of any SQL statement. Often, there are 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 related to the objects referenced and the conditions specified in the query. It can use either a cost-based or a rule-based approach.

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

You can influence the optimizer's choices by setting the optimizer approach and goal, and by gathering representative 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:

- "Choosing an Optimizer Approach and Goal" on page 1-10 for more information on optimization goals
- Chapter 3, "Gathering Optimizer Statistics" for more information on using statistics
- Chapter 5, "Optimizer Hints" for more information about using hints in SQL statements

Steps in 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 "How the Optimizer Performs Operations" on page 2-2.)
2	Statement transformation	For complex statements involving, for example, correlated subqueries or views, the optimizer might transform the original statement into an equivalent join statement. (See "How the Optimizer Transforms SQL Statements" on page 1-2.)
3	Choice of optimizer approaches	The optimizer chooses either a cost-based or rule-based approach and determines the goal of optimization. (See "Understanding Joins" on page 1-42.)

4	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 "Understanding Access Paths for the CBO" on page 1-23.)
5	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.
6	Choice of join methods	For any join statement, the optimizer chooses an operation to use to perform the join.

Understanding Execution Plans

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

See Also:

- "Understanding Access Paths for the CBO" on page 1-23
- "Understanding Access Paths for the RBO" on page 8-2

Overview of 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 explains a SQL statement that selects the name, job, salary, and department name for all employees whose salaries do not fall into a recommended salary range:

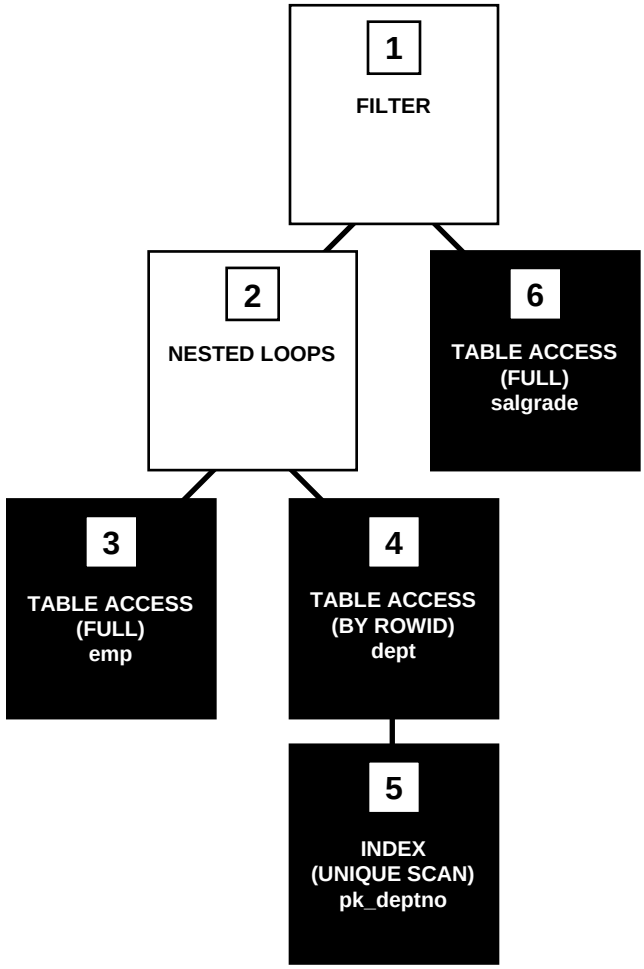
```
EXPLAIN PLAN FOR
SELECT ename, job, sal, dname
FROM emp, dept
WHERE emp.deptno = dept.deptno
AND NOT EXISTS
    (SELECT *
    FROM salgrade
    WHERE emp.sal BETWEEN losal AND hisal);
```

The following output table describes the statement explained 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
```

Figure 1–2 shows a graphical representation of the execution plan for this SQL statement.

Figure 1–2 An Execution Plan



Each box in Figure 1–2 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 1–2.

See Also:

- "Steps in the Execution Plan" on page 1-8
- Chapter 9, "Using EXPLAIN PLAN"

Steps in the 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 set*.

Figure 1–2 on page 1-7 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.

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:
  - 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 rows returned by the previous row source:
  - Step 2 performs the 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:

    - "Understanding Access Paths for the CBO" on page 1-23 and "Understanding Access Paths for the RBO" on page 8-2 for more information on access paths
    - "Understanding Joins" on page 1-42 for more information on the methods by which Oracle joins row sources

Understanding 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 1–2 on page 1-7). The rows returned by each step become the row sources of its parent step. Then, Oracle performs the parent steps.

> **Note:** The execution order in EXPLAIN PLAN begins with the line that is the furthest indented to the right. The next step is the parent of that line. If two lines are indented equally, then the top line is executed first.

For example, Oracle performs the following steps to execute the statement in Figure 1–2 on page 1-7:

- 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.

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 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*. This means that it chooses the least amount of resources necessary to process all rows accessed by the statement.

Also, Oracle can optimize a statement with the goal of best *response time*. This means that it uses the least amount of resources necessary to process the first row accessed by a SQL statement.

> **See Also:** "How the CBO Optimizes SQL Statements for Fast Response" on page 1-13

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 might return the entire query result faster, while the nested loops operation might 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. Usually, throughput is 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. Usually, response time is important in interactive applications, because the interactive user is waiting to see the first row or first few rows accessed by the statement.

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
- OPTIMIZER_GOAL Parameter of the ALTER SESSION Statement
- CBO Statistics in the Data Dictionary
- Optimizer SQL 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. 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 only some statistics, then the cost-based approach is used, and the optimizer must guess the statistics for the subjects without any statistics. This can result in suboptimal execution plans. 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.
`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_`*n*	The optimizer uses a cost-based approach, regardless of the presence of statistics, and optimizes with a goal of best response time to return first *n* number of rows. *N* can equal 1, 10, 100, or 1000.
`FIRST_ROWS`	The optimizer uses a mix of costs and heuristics to find a best plan for fast delivery of the first few rows. Note: The heuristic sometimes leads the CBO to generate a plan whose cost is significantly larger than the cost of a plan without applying the heuristic. `FIRST_ROWS` is available for backward compatibility and plan stability.
`RULE`	The optimizer chooses a rule-based approach for all SQL statements regardless of the presence of statistics.

For example: The following statement changes the goal of the CBO for the session to best response time:

```
ALTER SESSION SET OPTIMIZER_MODE = FIRST_ROWS_10;
```

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.

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 the user, including those issued by stored procedures and functions called during the session. It does not affect the optimization of internal recursive SQL statements that Oracle issues during the session for operations such as space management and data dictionary operations.

The OPTIMIZER_GOAL parameter can have the same values as the OPTIMIZER_MODE initialization parameter

> **See Also:** "OPTIMIZER_MODE Initialization Parameter" on page 1-11

Optimizer SQL Hints

A FIRST_ROWS(*n*), 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 the session, issue an ALTER SESSION SET OPTIMIZER_MODE statement with the ALL_ROWS, FIRST_ROWS, or FIRST_ROWS_*n* (where *n* = 1, 10, 100, or 1000) clause.
- To specify the goal of the CBO for an individual SQL statement, use the ALL_ROWS, FIRST_ROWS(*n*) (where *n* = any positive integer), or FIRST_ROWS hint.

> **See Also:** Chapter 5, "Optimizer Hints" for information on how to use hints

CBO Statistics in the Data Dictionary

The statistics used by the CBO are stored in the data dictionary. You can collect exact or estimated statistics about physical storage characteristics and data distribution in these schema objects by using the DBMS_STATS package or the ANALYZE statement.

To maintain the effectiveness of the CBO, you *must* have statistics that are representative of the data. It is possible to gather statistics on objects using either of the following:

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

For table columns that contain skewed data (in other words, values with large variations in number of duplicates), you should 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:** Chapter 3, "Gathering Optimizer Statistics" for detailed information on gathering statistics

How the CBO Optimizes SQL Statements for Fast Response

The CBO can optimize a SQL statement either for throughput or for fast response. Fast response optimization is used when the parameter OPTIMIZER_MODE is set to FIRST_ROWS_*n* (where *n* is 1, 10, 100, or 1000) or FIRST_ROWS. A hint FIRST_ROWS(*n*) (where *n* is any positive integer) or FIRST_ROWS can be used to optimize an individual SQL statement for fast response. Fast response optimization is suitable for online users, such as those using Oracle Forms or Web access. Typically, online users are interested in seeing the first few rows, and they seldom are interested in seeing the entire query result, especially when the result size is large. For such users, it makes sense to optimize the query to produce the first few rows as fast as possible, even if the time to produce the entire query result is not minimized.

With fast response optimization, the CBO generates a plan with lowest cost to produce the first row or the first few rows. The CBO employs two different fast response optimizations. The old method is used with the FIRST_ROWS hint or

parameter value. With the old method, the CBO uses a mixture of costs and rules to produce a plan. It is retained for backward compatibility reasons.

The new fast response optimization method is used when the FIRST_ROWS_*n* (where *n* can be 1, 10, 100, or 1000) parameter value is used or when the FIRST_ROWS(*n*) (where *n* can be any positive integer) hint is used. The new method is totally based on costs, and it is sensitive to the value of *n*. With small values of *n*, the CBO tends to generate plans that consist of nested loops joins with index lookups. With large values of *n*, the CBO tends to generate plans that consist of hash joins and full table scans.

The value of *n* should be chosen based on the online user requirement. It depends on how the result is displayed to the user. Generally, Oracle Forms users see the result one row at a time, and they typically are interested in seeing first few screens. Other online users see the result one group of rows at a time.

With the fast response method, the CBO explores different plans and, for each, computes the cost to produce the first *n* rows. It picks the plan that produces the first *N* rows with the lowest cost. Remember that with fast response optimization, a plan that produces the first *n* rows with the lowest cost might not be the optimal plan to produce the entire result. If the requirement is to obtain the entire result of a query, then fast response optimization should not be used. Instead use the ALL_ROWS parameter value or hint.

Features that Require the CBO

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

- Partitioned tables and indexes
- Index-organized tables
- Reverse key indexes
- Function-based indexes
- SAMPLE clauses in a SELECT statement
- Parallel query and parallel DML
- Star transformations and star joins
- Extensible optimizer
- Query rewrite with materialized views
- Enterprise Manager progress meter

- Hash joins
- Bitmap indexes and bitmap join indexes
- Index skip scans

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

Understanding the Cost-Based Optimizer (CBO)

In general, use the cost-based approach. Oracle Corporation is continually improving the CBO, and new features work only with the CBO. The rule-based approach is available for backward compatibility with legacy 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:** Chapter 5, "Optimizer Hints" for detailed information on hints

The CBO performs 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 access paths and join orders based on the estimated computer resources, including I/O, CPU, and memory.

   Serial plans with higher 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 lowest cost.

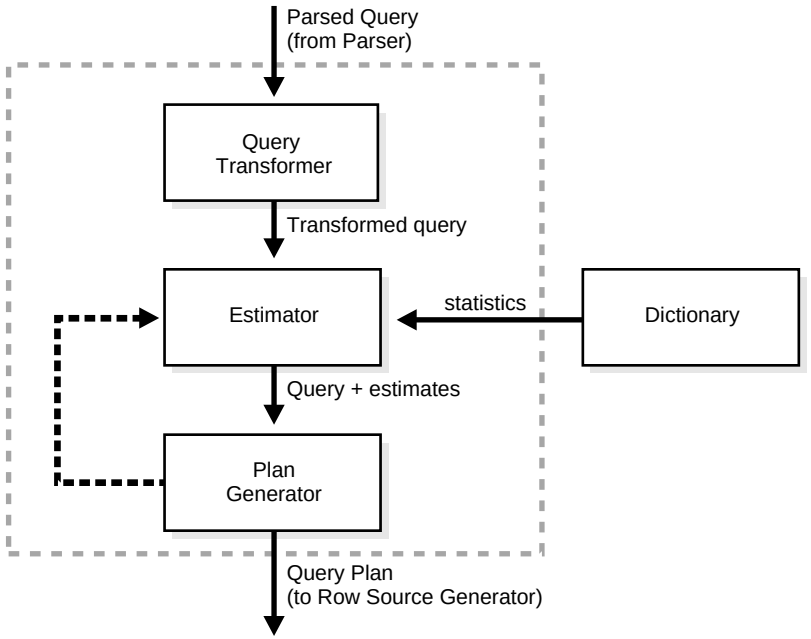
Architecture of the CBO

The CBO consists of the following three main components:

- Query Transformer
- Estimator
- Plan Generator

The CBO architecture is illustrated in Figure 1–3.

Figure 1–3 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. Four different query transformation techniques are employed by the query transformer: view

merging, predicate pushing, subquery unnesting, and query rewrite using materialized views. Any combination of these transformations might 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 suboptimal query plan, because the view is optimized separately from rest of the query.

The query transformer removes the potential suboptimal plan by merging the view query block into the query block that contains the view. Most types of views are merged. When a view is merged, the query block representing the view is merged into the containing query block. It is no longer necessary to generate a subplan, because view query block is eliminated.

Predicate Pushing For those views that are not merged, the query transformer can push the relevant predicates from the containing query block into the view query block. Doing so improves the subplan of the nonmerged view, because the pushed-in predicates can be used either to access indexes or can act as filters.

Subquery Unnesting Like a view, a subquery is 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 might 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 subqueries are unnested by the query transformer. For those subqueries that are not unnested, separate subplans are generated. To improve the execution speed of the overall query plan, the subplans are ordered in an efficient manner.

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 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:

- "How the Optimizer Transforms SQL Statements" on page 2-27
- *Oracle9i Data Warehousing Guide* for more information on query rewrite

Estimator

The estimator generates 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'`). The estimator makes this assumption because an equality predicate is expected to 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 estimates. Therefore, having histograms on columns that contain skewed data (in other words, values with large

variations in number of duplicates) greatly helps the CBO generate good selectivity estimates.

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 of 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 and the number of rows in the row set.

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 same row set of 100 rows is grouped by colx and coly, and the distinct cardinalities of colx and coly are 30 and 60 respectively, then the group cardinality lies between the maximum of the distinct cardinalities of colx and coly, and the lower of the product of the distinct cardinalities of colx and coly, and the number of rows in the row set.

This can be represented by the following formula:

```
group cardinality lies between max ( dist. card. colx
                                   , dist. card. coly )
                           and min ( (dist. card. colx * dist. card. coly)
                                   , num rows in row set )
```

Substituting the numbers from the example, the group cardinality is between the maximum of (30 and 60) and the minimum of (30*60 and 100). In other words, the group cardinality is between 60 and 100.

Cost The cost represents units of work or resource used. The CBO uses disk I/O, CPU usage, and memory usage as units of work. So, the cost used by the CBO represents an estimate of the number of disk I/Os and the amount of CPU and memory used 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 work units that are expected to be incurred when the query is executed and its result produced.

The **access path** 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.

Although the clustering factor is a property of the index, the clustering factor actually relates to the spread of similar indexed column values within data blocks in the table. A lower clustering factor indicates that the individual rows are concentrated within fewer blocks in the table. Conversely, a high clustering factor indicates that the individual rows are scattered more randomly across blocks in the table. Therefore, a high clustering factor means that it costs more to fetch rows by rowid using a range scan, because more blocks in the table need to be visited to return the data.

For example, assume the following:

- There is a table with 9 rows.
- There is a nonunique index on `col1` (in `tab1`).
- The `c1` column currently stores the values `A`, `B` and `C`.
- The table only has three Oracle blocks.

Example 1: The index clustering factor is low for the rows as they are arranged in the diagram below. This is because the rows that have the same indexed column values for `c1` are co-located within the same physical blocks in the table. This means that the cost of returning all of the rows that have the value A via a range scan is low, because only one block in the table needs to be read.

```
Block 1       Block 2       Block 3
-------       -------       --------
A  A  A       B  B  B       C  C  C
```

Example 2: If the same rows in the table are rearranged so that the index values are scattered across the table blocks (rather than colocated), then the index clustering factor is higher, because in order to retrieve all rows with the value A in `col1`, all three blocks in the table must be read.

```
Block 1       Block 2       Block 3
-------       -------       --------
A  B  C       A  B  C       A  B  C
```

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 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: "Understanding Joins" on page 1-42 for more information on joins

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 combinations 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 nested subqueries and nonmerged views. Each nested subquery or nonmerged 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 when finding 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 (in other words, 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.

Understanding Access Paths for the CBO

Access paths are ways in which data is retrieved from the database. For any row in any table accessed by a SQL statement, there are three common ways by which that row can be located and retrieved:

1. A row can be retrieved through a full table scan, which reads all rows from a table and filters out those that do not meet the selection criteria.
2. A row can be retrieved also through the use of an index, by traversing the index using the indexed column values specified by the statement.
3. A row can be retrieved by specifying the `ROWID`. `ROWID` access is the fastest way to retrieve a row, because this specifies the exact location of the row in the database.

In general, index access paths should be used for statements that retrieve a small subset of the table's rows, while full scans are more efficient when accessing a large portion of the table. OLTP systems, which consist of short-running SQL statements with high selectivity, often are characterized by the use of index access paths. Decision support systems, on the other hand, often use partitioned tables and perform full scans of the relevant partitions.

> **See Also:** "Understanding Access Paths for the RBO" on page 8-2 for the a list of the access paths that are available for the RBO, as well as their ranking

This section describes the following data access paths:

- Full Table Scans
- Sample Table Scans
- ROWID Scans
- Index Scans
- Cluster Scans
- Hash Scans

> **See Also:** Appendix A, "Schemas Used in Performance Examples"

Full Table Scans

During a full table scan, all blocks in the table below the high water mark are scanned. Each row is examined to determine whether it satisfies the statement's WHERE clause.

When Oracle performs a full tables scan, the blocks are read sequentially. Because the blocks are adjacent, I/O calls larger than a single block can be used to speed up the process. The size of the read calls range from one block to the number of blocks indicated by the initialization parameter DB_FILE_MULTIBLOCK_READ_COUNT. Using multiblock reads means a full table scan can be performed very efficiently. Each block is read only once.

For example:

```
SELECT last_name, first_name
  FROM per_people_f
 WHERE UPPER(full_name) LIKE :b1

Plan
-----------------------------------------------
SELECT STATEMENT
 TABLE ACCESS FULL PER_ALL_PEOPLE_F
```

The above example does a case-independent search for a name on a table containing all employees. Although the table might have several thousand employees, the number of rows for a given name ranges from 1 to 20. So, it might be better to use an index to access the desired rows. There is an index on the full_name in per_people_f_n54.

However, the optimizer is unable to use the index, because there is a function on the indexed column. Because it does not have any other access path, it uses a full table scan. If you need to use the index for case-independent searches, then function-based indexes created on search columns or mixed case data should not be allowed in the search columns.

When the Optimizer Uses Full Table Scans

The optimizer uses a full table scan if there is any of the following:

Lack of Access Paths If the query is unable to use any existing indexes, then it uses a full table scan.

Large Amount of Data If the optimizer thinks that the query will access a fraction of blocks in the table, then it uses a full table scan, even though there might be indexes available.

Small Table If a table contains less than DB_FILE_MULTIBLOCK_READ_COUNT blocks below the high water mark, which can be read in a single I/O call, then it could be cheaper to do a full table scan rather than an index range scan, regardless of the fraction of tables being accessed or indexes present.

Old Statistics If the table has not been analyzed since it was created, and if it has less than DB_FILE_MULTIBLOCK_READ_COUNT blocks below the high water mark, then the optimizer thinks that the table is small and uses a full table scan. Look at the LAST_ANALYZED and BLOCKS columns in the ALL_TABLES table to see what the statistics reflect.

High Degree of Parallelism A high degree of parallelism for the table skews the optimizer towards full table scans over range scans. Examine the DEGREE column in ALL_TABLES for the table to determine the degree of parallelism.

Full Table Scan Hints

Use the hint FULL(*table alias*).

Example before using the hint:

```
SELECT last_name, first_name
  FROM per_people_f
 WHERE full_name LIKE :b1;

Plan
-------------------------------------------------
SELECT STATEMENT
 TABLE ACCESS BY INDEX ROWID PER_PEOPLE_F
  INDEX RANGE SCAN PER_PEOPLE_F_N54
```

Example after using the hint:

```
SELECT /*+ FULL(f) */ last_name, first_name
  FROM per_people_f f
  WHERE full_name LIKE :b1;
```

```
Plan
-----------------------------------------------
SELECT STATEMENT
 TABLE ACCESS FULL PER_PEOPLE_F
```

Why a Full Table Scan is Faster for Accessing Large Amounts of Data

Full table scans are cheaper than index range scans when accessing a large fraction of the blocks in the table. This is because full table scans can use larger I/O calls, and fewer large I/O calls are cheaper than many small I/O calls.

The elapsed time for an I/O call consists of two components:

- Call setup time (head seek time, rotational latency)
- Data transfer time (this is typically ten MB/second or better)

In a typical I/O operation, setup costs consume most of the time. Data transfer time for an eight K buffer is less than one ms (out of the total time of ten ms). This means that you can transfer 128KB in about 20 ms with a single 128 KB call, opposed to 160 ms with 16 eight KB calls.

Example of Full Table Scan vs. Index Range Scan

In the following example, 20% of the blocks in a 10,000 block table are accessed. The following are true:

- `DB_FILE_MULTIBLOCK_READ_COUNT` = 16
- `DB_BLOCK_SIZE` =8k
- Number of 8K I/O calls required for index access = 2,000(table) + index
- Number of 128K I/O calls required for full table scan = 10,000/16 = 625

Assume that each 8K I/O takes 10 ms, and about 20 seconds are spent doing single block I/O for the table blocks. This does not include the additional time to perform the index block I/O. Assuming that each 128K I/O takes 20 ms, about 12.5 seconds are spent waiting for the data blocks, with no wait required for any index blocks.

The total time changes when CPU numbers are added in for crunching all the rows for a full table scan and for processing the index access in the range scan. But, the full table scan comes out faster with such a large fraction of blocks being accessed.

Assessing I/O for Blocks vs. Rows

Oracle does I/O by blocks. Therefore, the optimizer's decision to use full table scans is influenced by the percentage of blocks accessed, not rows. This is the index

clustering factor. If blocks contain single rows, then rows accessed and blocks accessed are the same.

However, most tables have multiple rows in each block. So, the desired number of rows could be clustered together in a few blocks, or they could be spread out over a larger number of blocks.

Consider a case where each block has 100 rows, but on an average only five rows per block meet the query condition. A query accessing 2% of the rows might need 40% of the blocks.

> **See Also:** "Estimator" on page 1-18 for more information on the index clustering factor

High Water Mark in DBA_TABLES

The data dictionary keeps track of the blocks that have been populated with rows. The high water mark is used as the end marker during a full table scan. The high water mark is stored in DBA_TABLES.BLOCKS. It is reset when the table is dropped or truncated.

For example, consider a table that had a large number of rows in the past. Most of the rows have been deleted, and now most of the blocks under the high water mark are empty. A full table scan on this table exhibits poor performance, because all the blocks below the high water mark are scanned.

Parallel Query Execution

When a full table scan is required, response time can be improved by using parallel slaves for scanning the table. Parallel query is used generally in low concurrency data warehousing environments due to the potential resource usage.

> **See Also:** *Oracle9i Data Warehousing Guide*

Sample Table Scans

A sample table scan retrieves a random sample of data from a table. This access path 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. To perform a sample table scan when sampling by blocks (the SAMPLE BLOCK clause), Oracle reads a specified percentage of the table's blocks.

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.

For 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
```

ROWID Scans

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.

ROWID via Index Example

In the following example, an index range scan is performed. The rowids retrieved are used to return the row data.

```
SELECT last_name, first_name
  FROM per_people_f
 WHERE full_name = :b1;

Plan
-----------------------------------------------
SELECT STATEMENT
 TABLE ACCESS BY INDEX ROWID PER_PEOPLE_F
  INDEX RANGE SCAN PER_PEOPLE_F_N54
```

When the Optimizer Uses ROWIDs

This is generally the second step after retrieving the ROWID from an index. The table access might be required for any columns in the statement not present in the index.

Access by ROWID does not need to follow every index scan. If the index contains all the columns needed for the statement, then table access by ROWID might not happen.

Note: ROWIDs are an internal Oracle representation of where data is stored. They can change between versions. Accessing data based on position is not recommended, because rows can move around due to row migration and chaining, and also after export and import. Foreign keys should be based on primary keys. For more information on ROWIDs, see *Oracle9i Application Developer's Guide - Fundamentals*.

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:

- Index Unique Scans
- Index Range Scans
- Index Range Scans Descending
- Index Skip Scans
- Full Scans
- Fast Full Index Scans
- Index Joins
- Bitmap Joins

Index Unique Scans

This returns, at most, a single rowid. 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.

Index Unique Scan Example The example below queries the orders table so_headers for a given order (header_id):

```
SELECT attribute2
  FROM so_headers
 WHERE order_number = :b1
   AND order_Type_id = :b2;

Plan
-------------------------------------------------
SELECT STATEMENT
 TABLE ACCESS BY ROWID OF SO_HEADERS_ALL
  INDEX UNIQUE SCAN  OF SO_HEADERS_U2
```

When the Optimizer Uses Index Unique Scans This access path is used when all columns of a unique (B-tree) index are specified with equality conditions.

Example that does not use Unique Scans with a Unique Index None of the statements below use a UNIQUE scan, because the complete unique key is not being used in equality condition.

```
SELECT attribute2
  FROM so_headers
 WHERE order_number = :b1
   AND order_type_id > 0;

Plan
-------------------------------------------------
SELECT STATEMENT
 TABLE ACCESS BY ROWID  OF SO_HEADERS_ALL
  INDEX RANGE SCAN  OF SO_HEADERS_U2
```

> **See Also:** *Oracle9i Database Concepts* for more details on index structures and for detailed information on how a B-tree is searched

Index Unique Scan Hints In general, you should not need to give a hint to do a unique scan. There might be cases where the table is across a database link and being

driven into from a local table, or where the table is small enough for the optimizer to prefer a full table scan.

The hint INDEX(*alias index_name*) specifies the index to use, but not a specific access path (range scan or unique scan).

Index Range Scans

Index range scan is a common operation for accessing selective data. It can be bounded (bounded on both sides) or unbounded (on one or both sides). Data is returned in the ascending order of index columns. Multiple rows with identical values are sorted in ascending order by the ROWIDs.

If data is required to be sorted by order, then use the ORDER BY clause, and do not rely on an index. If an index can be used to satisfy an ORDER BY clause, then the optimizer uses this and avoids a sort.

Index Range Scan Example In the example below, the order has been imported from a legacy system, and you are querying the order by the number in the legacy system. This should be a highly selective query, and you see the query using the index on the column to retrieve the desired rows. The data returned is sorted in ascending order by the original_system_reference, ROWIDs. Because the index column original_system_reference is identical for the selected rows here, the data is sorted by the ROWIDs.

```
SELECT attribute2, header_id
  FROM so_headers
 WHERE original_system_reference = :b1;

Plan
-------------------------------------------------
SELECT STATEMENT
 TABLE ACCESS BY ROWID  OF SO_HEADERS_ALL
  INDEX RANGE SCAN  OF SO_HEADERS_N5
```

When the Optimizer Uses Index Range Scans The optimizer uses a range scan when the optimizer finds one or more leading columns of an index in conditions, such as the following:

- col1 = :b1
- col1 < :b1
- col1 > :b1

- Wild-card searches on character columns. (Wild-card searches should not be in a leading position. The condition `col1` like '%ASD' does not result in a range scan.)
- `AND` combination of the above for leading columns in the index

Range scans can use unique or nonunique indexes. Range scans avoid sorting when index columns constitute the `ORDER BY` / `GROUP BY` clause.

Index Range Scan Hints A hint might be required if the optimizer chooses some other index or uses a full table scan. The hint `INDEX`(*table_alias index_name*) specifies the index to use.

In the example below, the column `s2` has a skewed distribution:

```
    S2       Rows     Blocks
------ ---------- ----------
     0        769        347
     4       1460        881
     5          5          4
     8        816        590
    18  1,028,946    343,043
```

The column has histograms, so the optimizer knows about the distribution. However, with a bind variable, the optimizer does not know the value and could choose a full table scan. Therefore, there are two options

1. Use literals rather than bind variables, which can cause problems due to nonsharing of the SQL statements.
2. Use hints in order to share the statements.

Example before using the hint:

```
SELECT l.line_id, l.revenue_amount
  FROM so_lines_all l
 WHERE l.s2 = :b1;

Plan
-----------------------------------------------
SELECT STATEMENT
 TABLE ACCESS FULL SO_LINES_ALL
```

Example using literal values:

```
SELECT l.line_id, l.revenue_amount
  FROM so_lines_all l
 WHERE l.s2 = 4;

Plan
-------------------------------------------------
SELECT STATEMENT
 TABLE ACCESS BY INDEX ROWID SO_LINES_ALL
  INDEX RANGE SCAN SO_LINES_N7
```

Example using bind variables and hint:

```
SELECT /*+ INDEX(l so_lines_n7) */ l.line_id, l.revenue_amount
  FROM so_lines_all l
 WHERE l.s2 = :b1;

Plan
-------------------------------------------------
SELECT STATEMENT
 TABLE ACCESS BY INDEX ROWID SO_LINES_ALL
  INDEX RANGE SCAN SO_LINES_N7
```

Index Range Scans Descending

Index range scan descending is identical to index range scan, except that the data is returned in a descending order. (Indexes, by default, are stored in ascending order.) Usually, this is used when ordering data in a descending order to return the most recent data first, or when seeking a value less than some value.

Index Range Scan Descending Examples In the example below, the index used is the two column unique index on `order_number`, `order_type_id`. So, the data is sorted in descending order by the `order_number`, `order_type_id`, `rowid` of the selected rows. However, because there is only one row per `order_number`, `order_type_id` (because `so_headers_u2` is a unique index on the two columns), the rows are sorted by `order_number`, `order_type_id`.

```
SELECT attribute2, header_id
  FROM so_headers_all a
 WHERE order_number = :b1
 ORDER BY order_number DESC;

Plan
-----------------------------------------------
SELECT STATEMENT
 TABLE ACCESS BY ROWID  OF SO_HEADERS_ALL
  INDEX RANGE SCAN DESCENDING SO_HEADERS_U2
```

In the next example below, the index used is a single column index on `purchase_order_num`. The data is retrieved in descending order by `purchase_order_num`, `rowid`.

```
SELECT attribute2, header_id
  FROM so_headers_all a
 WHERE purchase_order_num BETWEEN :b1 AND :b2
 ORDER BY purchase_order_num DESC;

Plan
-----------------------------------------------
SELECT STATEMENT
 TABLE ACCESS BY INDEX ROWID SO_HEADERS_ALL
  INDEX RANGE SCAN DESCENDING SO_HEADERS_N3
```

When the Optimizer Uses Index Range Scans Descending The optimizer uses index range scan descending when an order by descending clause can be satisfied by an index.

Index Range Scan Descending Hints The hint `INDEX_DESC(`*table_alias index_name*`)` is used for this access path.

Example before using hint:

```
SELECT a.original_system_reference, a.original_system_source_code, a.header_id
  FROM so_headers_all a
 WHERE a.original_system_reference = :b1;

Plan
-----------------------------------------------
SELECT STATEMENT
 TABLE ACCESS BY INDEX ROWID SO_HEADERS_ALL
  INDEX RANGE SCAN SO_HEADERS_N5
```

Example after using hint #1:

```
SELECT /*+INDEX_DESC(a so_headers_n5) */
       a.original_system_reference, a.original_system_source_code, a.header_id
  FROM so_headers_all a
 WHERE a.original_system_reference = :b1;
```

The data with this hint is sorted in descending order by `original_system_reference`, `original_system_code`, `rowid`.

```
Plan
-------------------------------------------------
SELECT STATEMENT
 TABLE ACCESS BY INDEX ROWID SO_HEADERS_ALL
  INDEX RANGE SCAN DESCENDING SO_HEADERS_N5
```

Example after using hint #2:

```
SELECT /*+INDEX_DESC(a so_headers_n9) */
       a.original_system_reference, a.original_system_source_code, a.header_id
  FROM so_headers_all a
 WHERE a.original_system_reference = :b1;
```

The data with this hint is sorted in descending order by `original_system_reference`, `rowid`.

```
Plan
-------------------------------------------------
SELECT STATEMENT
 TABLE ACCESS BY INDEX ROWID SO_HEADERS_ALL
  INDEX RANGE SCAN DESCENDING SO_HEADERS_N9
```

Index Skip Scans

Index skip scans improve index scans by nonprefix columns. Often, it is faster to scan index blocks than it is to scan table data blocks.

Skip scanning lets a composite index be logically split into smaller subindexes. For example, table `emp` (`sex`, `empno`, `address`) with a composite index on (`sex`, `empno`). The number of logical subindexes is determined by the number of distinct values in the initial column.

Skip scanning is useful when the initial column of the composite index is not specified in the query. In other words, it is "skipped."

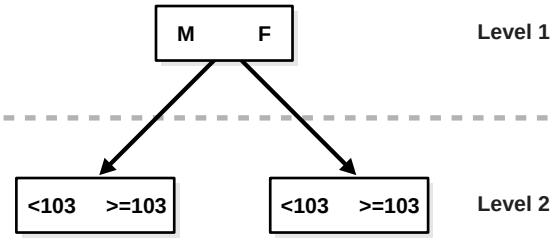
For example above, suppose you have the following index data:

```
('F',98)
('F',100)
('F',102)
('F',104)
('M',101)
('M',103)
('M',105)
```

Skip scanning is advantageous if there are few distinct values of the leading column of the composite index and many values of the nonleading key of the index.

The index is split logically in the following two subindexes:

1. The first one has the keys with the value `'F'`.
2. The second one has all the values with the value `'M'`.



In the following query,

```
SELECT *
FROM emp
WHERE empno = 101;
```

the column `sex` is skipped. A complete scan of the index is not performed, but the subindex with the value `'F'` is searched first, then the subindex with the value `'M'` is searched.

Full Scans

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. This can be used to eliminate a sort operation, because the data is ordered by the index key. It reads the blocks singly.

Fast Full Index Scans

Fast full index scans are 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, because the data is *not* ordered by the index key. 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.

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. The following query and plan illustrate this feature.

```
SELECT COUNT(*)
FROM t1, t2
WHERE t1.c1 > 50
     AND t1.c2 = t2.c1;

Plan
-------------------------------------------------

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.

Fast Full Index Scan Restrictions Fast full index scans have the following restrictions:

- At least one indexed column of the table must have the NOT NULL constraint.
- 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 might decide not to use it.

Fast Full Index Scan Hints Fast full scan has a special index hint, INDEX_FFS, which has the same format and arguments as the regular INDEX hint.

> **See Also:** Chapter 5, "Optimizer Hints" for detailed information on the INDEX_FFS hint

Index Joins

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.

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;

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

Index Join Hints You can specify it with the initialization parameter OPTIMIZER_FEATURES_ENABLE or the INDEX_JOIN hint.

Bitmap Joins

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 and bitmap join indexes are available only if you have purchased the Oracle9*i* Enterprise Edition. For more information on bitmap indexes, see *Oracle9i Data Warehousing Guide*.

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.

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:** Chapter 5, "Optimizer Hints" for information about hints in SQL statements

The following examples illustrate how the optimizer uses selectivity.

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 the following 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 distributed uniformly, 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 distributed evenly 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 might 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 can be especially apparent, the optimizer might 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 )
```

Understanding Joins

Joins are statements that retrieve data from more than one table. A join is characterized by multiple tables in the FROM clause, and the relationship between the tables is defined through the existence of a join condition in the WHERE clause.

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.

See Also: Appendix A, "Schemas Used in Performance Examples"

How the CBO Executes 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 "Understanding Access Paths for the RBO" on page 8-2 and "Understanding Access Paths for the CBO" on page 1-23.)
Join Method	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.

How the CBO 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 might 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 presorted. In the presorted 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.

Join methods include:

- Nested Loop Joins
- Nested Loop Outer Joins
- Hash Joins
- Hash Join Outer Joins
- Sort Merge Joins
- Sort Merge Outer Joins
- Cartesian Joins
- Full Outer Joins

Nested Loop Joins

Nested loop (NL) joins are useful for joining small subsets of data and if the join condition is an efficient way of accessing the second table.

It is very important to ensure that the inner table is driven from the outer table. If the inner table's access path is independent of the outer table, then the same rows are retrieved for every iteration of the outer loop. This can degrade performance considerably. In such cases, hash joins joining the two independent row sources perform better.

> **See Also:** "Cartesian Joins" on page 1-57

For nested loop joins:

1. The optimizer determines a driving table (the outer loop).
2. The other table is designated the inner table.
3. For every row in the first (outer) table, Oracle accesses all the rows in the second (inner) table. The outer loop is for every row in first table and the inner loop is for every row in second table. The outer loop appears above the inner loop in the execution plan. For example:

```
NESTED LOOP
  <Outer Loop>
  <Inner Loop>
```

Nested Loop Join Example

```
SELECT a.selling_price*a.ordered_quantity
  FROM so_lines a,so_headers b
 WHERE b.customer_id = :b1
   AND a.header_id = b.header_id

Plan
-------------------------------------------------
SELECT STATEMENT
 NESTED LOOPS
  TABLE ACCESS BY INDEX ROWID SO_HEADERS_ALL
   INDEX RANGE SCAN SO_HEADERS_N1
  TABLE ACCESS BY INDEX ROWID SO_LINES_ALL
   INDEX RANGE SCAN SO_LINES_N1
```

Outer loop From the execution plan, the outer loop and the equivalent statement are

```
TABLE ACCESS BY INDEX ROWID SO_HEADERS_ALL
 INDEX RANGE SCAN SO_HEADERS_N1 (CUSTOMER_ID)

SELECT <some columns>
  FROM so_headers b
 WHERE b.customer_id = :b1
```

Inner loop The execution plan shows that the inner loop being iterated for every row fetched from the outer loop is

```
TABLE ACCESS BY INDEX ROWID SO_LINES_ALL
 INDEX RANGE SCAN SO_LINES_N1 (HEADER_ID)

SELECT <some columns>
  FROM so_lines a
WHERE a.header_id = :b2
```

Complete statement Therefore, the statement that retrieves the lines for a customer can be broken down into two loops:

- **Outer loop** - Retrieve all the orders for the given customer
- **Inner loop** - For every order retrieved in step #1, retrieve the lines

When the Optimizer Uses Nested Loop Joins

The optimizer uses NL joins when joining small number of rows with a good driving condition between the two tables. This is a join in which you drive from the

outer loop to the inner loop, so the order of tables in the execution plan is important.

The outer loop is the driving row source. It produces a set of rows for driving the join condition. The row source can be a table accessed using an index scan or a full table scan. Also, the rows can be produced from any other operation. For example, the output from a nested loop join can be used as a row source for another nested loop join.

The inner loop is iterated for every row returned from the outer loop. This should ideally be an index scan. If the access path for the inner loop is not dependent on the outer loop, then you can end up with a Cartesian product. In such a case, for every iteration of the outer loop, the inner loop produces the same set of rows. Hence, it is better to use other join methods where two independent row sources are joined together.

Nested Loop Join Hints

If the optimizer is choosing to use some other join method, use the USE_NL(a b) hint, where a and b are the aliases of the tables being joined.

In the example below, data is small enough for the optimizer to prefer full table scans and use hash joins.

```
SELECT l.selling_price*l.ordered_quantity
  FROM so_lines_all l, so_headers_all h
 WHERE h.customer_id = :b1
   AND l.header_id = h.header_id;

Plan
-------------------------------------------------
SELECT STATEMENT
 HASH JOIN
  TABLE ACCESS FULL SO_HEADERS_ALL
  TABLE ACCESS FULL SO_LINES_ALL
```

The following gives a hint that changes the join method to nested loop. In this statement, so_headers_all is accessed via a full table scan and the filter condition customer_id = :b1 is applied to every row. For every row that meets the filter condition, so_lines_all is accessed via the index so_lines_n1 (header_id).

```
SELECT /*+ USE_NL(l h) */ l.selling_price*l.ordered_quantity
  FROM so_lines_all l, so_headers_all h
 WHERE h.customer_id = :b1
   AND l.header_id = h.header_id;

Plan
-------------------------------------------------
SELECT STATEMENT
 NESTED LOOPS
  TABLE ACCESS FULL SO_HEADERS_ALL
  TABLE ACCESS BY INDEX ROWID SO_LINES_ALL
   INDEX RANGE SCAN SO_LINES_N1
```

Adding another hint here avoids the full table scan on `so_headers_all` to get an execution plan similar to that on the larger systems (even though it might not be particularly efficient here).

```
SELECT /*+ USE_NL(l h) INDEX(h so_headers_n1) */ l.selling_price*l.ordered_
quantity
  FROM so_lines_all l, so_headers_all h
 WHERE h.customer_id = :b1
   AND l.header_id = h.header_id;

Plan
-------------------------------------------------
SELECT STATEMENT
 NESTED LOOPS
  TABLE ACCESS BY INDEX ROWID SO_HEADERS_ALL
   INDEX RANGE SCAN SO_HEADERS_N1
  TABLE ACCESS BY INDEX ROWID SO_LINES_ALL
   INDEX RANGE SCAN SO_LINES_N1
```

Nesting Nested Loops

Because the outer loop can be a nested loop, you can nest them together to join more tables as needed, where each loop is a data access method.

```
SELECT STATEMENT
 NESTED LOOP 3
  NESTED LOOP 2          (OUTER LOOP 3.1)
   NESTED LOOP 1         (OUTER LOOP 2.1)
    OUTER LOOP 1.1     - #1
    INNER LOOP 1.2     - #2
   INNER LOOP 2.2      - #3
  INNER LOOP 3.2       - #4
```

Nested Loop Outer Joins

This operation is used when an outer join is used between two tables. The outer join returns the outer (preserved) table rows, even when there are no corresponding rows in the inner (optional) table.

In a regular outer join, the optimizer chooses the order of tables (driving and driven) based on the cost. However, in an outer join, the order of tables is determined by the join condition. The outer table (whose rows are being preserved) is used to drive to the inner table.

Nested Loop Outer Join Example

In the example below, all the customers created in the last 30 days are queried to see how active they are. An outer join is needed so that you don't miss the customers who do not have any orders.

```
SELECT customer_name, 86(nvl2(h.customer_id,0,1)) "Count"
  FROM ra_customers c, so_headers_all h
 WHERE c.creation_date > SYSDATE - 30
   AND c.customer_id = h.customer_id(+)
 GROUP BY customer_name;

Plan
-------------------------------------------------
SELECT STATEMENT
 SORT GROUP BY
  NESTED LOOPS OUTER
   TABLE ACCESS BY INDEX ROWID RA_CUSTOMERS
    INDEX RANGE SCAN RA_CUSTOMERS_N2
   INDEX RANGE SCAN SO_HEADERS_N1
```

In this case, the outer join condition is the following

```
ra_customers.customer_id = so_headers_all.customer_id(+)
```

This means the following:

- The outer table is `ra_customers`.
- The inner table is `so_headers_all`.
- The join preserves the `ra_customers` rows, including those rows without a corresponding row in `so_headers_all`.
- The join can drive only from `ra_customers` to `so_headers_all`.

When the Optimizer Uses Nested Loop Outer Joins

The optimizer uses nested loop joins to process an outer join if the following are true:

- It is possible to drive from the outer table to inner table.
- Data volume is low enough to make nested loop method efficient.

Nested Loop Outer Join Hints

To have the optimizer to use a nested loop for the join, use the hint USE_NL.

Hash Joins

Hash joins are used for joining large data sets. The optimizer uses the smaller of the two tables / data sources to build a hash table on the join key in memory. It then scans the larger table, probing the hash table to find the joined rows.

This is best when the smaller table is small enough to fit in available memory. The cost is then limited to a single read pass over the data for the two tables.

However, if the hash table grows too big to fit into the memory, then the optimizer breaks it up into different partitions. As the partitions exceed allocated memory, parts are written to disk to temporary segments.

After the hash table is complete:

- Table 2 is scanned.
- It is broken up into partitions like table 1.
- Partitions are written to disk.

When the hash table build completes, it is possible that an entire hash table partition is resident in memory. Then, you do not need to build the corresponding

partition for the second table. When table 2 is scanned, rows that hash to the resident hash table partition can be joined and returned immediately.

Now, each hash table partition is read into memory:

- The corresponding table 2 partition is scanned.
- The hash table is probed to return the joined rows.

This is repeated for the rest of the partitions. The cost can increase to two read passes over the data and one write pass over the data.

There is also the possibility that if the hash table does not fit in the memory, then parts of it must be swapped in and out, depending on the rows retrieved from the second table. Performance for this scenario can be extremely poor.

When the Optimizer Uses Hash Joins

The optimizer uses a hash join to join two tables if they are joined using an equijoin and if either of the following are true:

- A large amount of data needs to be joined.
- A large fraction of the table needs to be joined.

Hash Joins Example

```
SELECT h.customer_id, l.selling_price*l.ordered_quantity
  FROM so_headers_all h ,so_lines_all l
 WHERE l.header_id = h.header_id

Plan
-----------------------------------------------
SELECT STATEMENT
 HASH JOIN
  TABLE ACCESS FULL SO_HEADERS_ALL
  TABLE ACCESS FULL SO_LINES_ALL
```

In the above example, the table `so_headers_all` is used to build the hash table, and `so_lines_all` is the larger table, which is scanned later.

Hash Join Hints

Use the `USE_HASH` hint to advise the optimizer to use a hash join when joining two tables together. Investigate the values for the parameters `HASH_AREA_SIZE` and `HASH_JOIN_ENABLED` if you are having trouble getting the optimizer to use hash joins.

Before Hint

```
SELECT l.promise_date,l.inventory_item_id, SUM(l2.ordered_quantity)
  FROM so_lines_all l, so_lines_all l2
 WHERE l.inventory_item_id = l2.inventory_item_id
   AND l.warehouse_id = l2.warehouse_id
   AND l2.promise_date < l.promise_date
GROUP BY l.inventory_item_id, l.promise_date, l.header_id

Plan
-------------------------------------------------
SELECT STATEMENT
 SORT GROUP BY
  MERGE JOIN
   SORT JOIN
    TABLE ACCESS FULL SO_LINES_ALL
   FILTER
    SORT JOIN
     TABLE ACCESS FULL SO_LINES_ALL
```

After Hint

```
SELECT /*+USE_HASH(l l2) */ l.promise_date,l.inventory_item_id, SUM(l2.ordered_
quantity)
  FROM so_lines_all l, so_lines_all l2
 WHERE l.inventory_item_id = l2.inventory_item_id
   AND l.warehouse_id = l2.warehouse_id
   AND l2.promise_date < l.promise_date
GROUP BY l.inventory_item_id, l.promise_date, l.header_id

Plan
-------------------------------------------------
SELECT STATEMENT
 SORT GROUP BY
  HASH JOIN
   TABLE ACCESS FULL SO_LINES_ALL
   TABLE ACCESS FULL SO_LINES_ALL
```

Hash Join Outer Joins

This operation is used for outer joins where the optimizer decides that the amount of data is large enough to warrant an hash join, or it is unable to drive from the outer table to the inner table.

Like an outer join, the order of tables is not determined by the cost, but by the join condition. The outer table (whose rows are being preserved) is used to build the hash table, and the inner table is used to probe the hash table.

The example below looks for inactive customers (more than a year old and have no orders). An outer join returns NULL for the inner table columns along with the outer (preserved) table rows when it does not find any corresponding rows in the inner table. This finds all the ra_customers rows that do not have any so_headers_all rows.

```
SELECT customer_name
  FROM ra_customers c, so_headers_all h
 WHERE c.creation_date < SYSDATE - 365
   AND h.customer_id(+) = c.customer_id
   AND h.customer_id IS NULL;

Plan
-------------------------------------------------
SELECT STATEMENT
 FILTER
  HASH JOIN OUTER
   TABLE ACCESS BY INDEX ROWID RA_CUSTOMERS
    INDEX RANGE SCAN RA_CUSTOMERS_N2
   INDEX FAST FULL SCAN SO_HEADERS_N1
```

In this case, the outer join condition is the following:

```
ra_customers.customer_id = so_headers_all.customer_id(+)
```

This means the following:

- The outer table is ra_customers.
- The inner table is so_headers_all.
- The join preserves the ra_customers rows, including those rows without a corresponding row in so_headers_all.
- The hash table is built using ra_customers.
- The hash table is probed using so_headers_all.

You could have used a NOT EXISTS subquery to return the rows, but because you are querying all the rows in the table, the hash join performs better (unless the NOT EXISTS subquery is un-nested).

When the Optimizer Uses Hash Join Outer Joins

The optimizer uses hash joins for processing an outer join if the data volume is high enough to make hash join method efficient or if it is not possible to drive from the outer table to inner table.

In the following example, the outer join is to a multitable view. The optimizer cannot drive into the view like in a normal join or push the predicates, so it builds the entire row set of the view.

```
SELECT c.customer_name, sum(revenue)
  FROM ra_customers c, orders h
 WHERE c.creation_date > sysdate - 30
   AND h.customer_id(+) = c.customer_id
 GROUP BY c.customer_name;

Plan
-------------------------------------------------
SELECT STATEMENT
 SORT GROUP BY
  HASH JOIN OUTER
   TABLE ACCESS BY INDEX ROWID RA_CUSTOMERS
    INDEX RANGE SCAN RA_CUSTOMERS_N2
   VIEW  ORDERS
    SORT GROUP BY
     HASH JOIN
      TABLE ACCESS FULL SO_HEADERS_ALL
      TABLE ACCESS FULL SO_LINES_ALL
```

View Definition

```
CREATE OR REPLACE view orders AS
SELECT h.order_number, SUM(l.revenue_amount) revenue, h.header_id, h.customer_id
  FROM so_headers_All h, so_lines_all l
 WHERE h.header_id = l.header_id
 GROUP BY h.order_number, h.header_id, h.customer_id;
```

Hash Join Outer Hints

To use hash join for doing the join, use the hint USE_HASH.

Sort Merge Joins

Sort merge joins can be used to join rows from two independent sources. Hash joins generally perform better than sort merge joins. However, sort merge joins can perform better than hash joins if the row sources are sorted already, and if a sort

operation does not have to be done. However, if this involves choosing a slower access method (index scan vs. full table scan), then the benefit of using a sort merge might be lost.

Sort merge joins are useful when the join condition between two tables is an inequality condition (but not a nonequality) like <, <=, >, or >=. Sort merge joins perform better than nested loop joins for large data sets. (You cannot use hash joins unless there is an equality condition).

In a merge join, there is no concept of a driving table.

- Both the inputs are sorted on the join key - sort join operation
- The sorted lists are merged together - merge join operation

If the input is sorted already by the join column, then there is not a sort join operation for that row source.

Sort Merge Join Example

```
SELECT SUM(l.revenue_amount), l2.creation_date
  FROM so_lines_all l, so_lines_all l2
 WHERE l.creation_date < l2.creation_date
   AND l.header_id <> l2.header_id
GROUP BY l2.creation_date, l2.line_id

Plan
-------------------------------------------------
SELECT STATEMENT
 SORT GROUP BY
  MERGE JOIN
   SORT JOIN
    TABLE ACCESS FULL SO_LINES_ALL
   FILTER
    SORT JOIN
     TABLE ACCESS FULL SO_LINES_ALL
```

In the above example tracking revenue generation, there is no equi-join operation. Therefore, you can use either nested loops or sort merge to join the data. With the data volumes, sort merge (as chosen by the optimizer) is the better option.

When the Optimizer Uses Sort Merge Joins

The optimizer can choose sort merge join over hash join for joining large amounts of data if any of the following are true:

- The join condition between two tables is not an equi-join
- OPTIMIZER_MODE is set to RULE
- HASH_JOIN_ENABLED is false
- Because of sorts already required by other operations, the optimizer finds it is cheaper to use sort merge over hash join
- The optimizer thinks that the cost of hash join is higher based on the settings of HASH_AREA_SIZE and SORT_AREA_SIZE

In this following example tracking inventory consumption, the optimizer avoids a sort for the GROUP BY if it chooses the sort merge operation.

```
SELECT msi.inventory_item_id, SUM(l.ordered_quantity)
  FROM mtl_system_items msi, so_lines_all l
 WHERE msi.inventory_item_id = l.inventory_item_id
   AND msi.inventory_item_id BETWEEN :b1 AND :b2
   GROUP BY msi.inventory_item_id

Plan
-------------------------------------------------
SELECT STATEMENT
 SORT GROUP BY NOSORT
  MERGE JOIN
   TABLE ACCESS BY INDEX ROWID SO_LINES_ALL
    INDEX RANGE SCAN SO_LINES_N5
   SORT JOIN
    INDEX RANGE SCAN MTL_SYSTEM_ITEMS_U1
```

Sort Merge Join Hints

Use the USE_MERGE hint to advise the optimizer to use a merge join when joining the two tables together. In addition, it might be necessary to give hints to force an access path. For example:

```
SELECT h.customer_id, l.selling_price*l.ordered_quantity
  FROM so_headers_all h ,so_lines_all l
 WHERE l.header_id = h.header_id
```

```
Plan
-----------------------------------------------
SELECT STATEMENT
 NESTED LOOPS
  TABLE ACCESS FULL SO_LINES_ALL
  TABLE ACCESS BY INDEX ROWID SO_HEADERS_ALL
   INDEX UNIQUE SCAN SO_HEADERS_U1
```

The optimizer chooses a full scan on `so_lines_all`, thus avoiding a sort. However, there is an increased cost because now a large table is accessed via an index and single block reads opposed to faster access via full table scan.

```
SELECT /*+USE_MERGE(h l) */ h.customer_id, l.selling_price*l.ordered_quantity
  FROM so_headers_all h ,so_lines_all l
 WHERE l.header_id = h.header_id

Plan
-----------------------------------------------
SELECT STATEMENT
 MERGE JOIN
  TABLE ACCESS BY INDEX ROWID SO_LINES_ALL
   INDEX FULL SCAN SO_LINES_N1
  SORT JOIN
   TABLE ACCESS FULL SO_HEADERS_ALL
```

Here, there is a full scan on both tables and the resulting inputs are sorted before merging them:

```
SELECT /*+USE_MERGE(h l) FULL(l) */ h.customer_id, l.selling_price*l.ordered_
quantity
  FROM so_headers_all h ,so_lines_all l
 WHERE l.header_id = h.header_id

Plan
-----------------------------------------------
SELECT STATEMENT
 MERGE JOIN
  SORT JOIN
   TABLE ACCESS FULL SO_HEADERS_ALL
  SORT JOIN
   TABLE ACCESS FULL SO_LINES_ALL
```

Sort Merge Outer Joins

When an outer join cannot drive from the outer (preserved) table to the inner (optional) table, it cannot use hash join or nested loop joins. Then it uses the sort merge join for executing the join operation.

When the Optimizer Uses Sort Merge Outer Joins

The optimizer uses sort merge for an outer join if a nested loop join is inefficient. A nested loop join can be inefficient because of data volumes, or because of sorts already required by other operations, the optimizer finds it is cheaper to use a sort merge over a hash join.

In the following example of an inventory usage report, the optimizer avoids a sort for the GROUP BY operation by using the sort merge operation.

```
SELECT msi.inventory_item_id, SUM(l.ordered_quantity)
  FROM mtl_system_items msi, so_lines_all l
 WHERE msi.inventory_item_id = l.inventory_item_id(+)
   GROUP BY msi.inventory_item_id;

Plan
-------------------------------------------------
SELECT STATEMENT
 SORT GROUP BY NOSORT
  MERGE JOIN OUTER
   SORT JOIN
    INDEX FAST FULL SCAN MTL_SYSTEM_ITEMS_U1
   SORT JOIN
    TABLE ACCESS FULL SO_LINES_ALL
```

Sort Merge Outer Join Hints

To use sort merge for doing the join, use the hint USE_MERGE.

Cartesian Joins

A Cartesian join happens when one or more of the tables does not have any join conditions to any other tables in the statement. The optimizer joins every row from one data source with every row from the other data source (Cartesian product of the two sets).

In some cases, there might be a common filter condition between the two tables that could be picked up by the optimizer as a possible join condition. This is even more

dangerous, because the joins are not flagged in the execution plan as being a Cartesian product.

Cartesian Join Examples

Cartesian joins generally result from poorly-written SQL. The example below has three tables in the FROM clause, but only one join condition joining the two tables.

Although this is a simplified version with three tables, this can happen when someone is writing a query involving large number of tables and an extra table gets into the FROM clause but not into the WHERE clause. With such queries, a DISTINCT clause can weed out multiple rows.

> **Note:** This example is not the recommended method. You should specify join criteria, thereby avoiding the Cartesian product.
>
> In this example, DISTINCT removes the extra tuples generated by the Cartesian product, but the performance is severely degraded.

```
SELECT DISTINCT h.header_id, l.line_id, l.ordered_quantity
  FROM so_lines_all l, so_headers_all h, so_lines_all l2
 WHERE h.customer_id = :b1
   AND l.header_id = h.header_id;

Plan
-------------------------------------------------
SELECT STATEMENT
 SORT UNIQUE
  MERGE JOIN CARTESIAN
   NESTED LOOPS
    TABLE ACCESS BY INDEX ROWID SO_HEADERS_ALL
     INDEX RANGE SCAN SO_HEADERS_N1
    TABLE ACCESS BY INDEX ROWID SO_LINES_ALL
     INDEX RANGE SCAN SO_LINES_N1
   SORT JOIN
    INDEX FAST FULL SCAN SO_LINES_N1
```

The optimizer gets the ROWIDs for so_lines_all from the index so_lines_n1. This is because header_id is a NOT NULL column, so the index has all the ROWIDs. It sorts the results by the join key.

The optimizer joins L and H using a nested loop join, and for every row returned, it returns all the rows from the entire so_lines_all table. The execution plan shows that a merge join is used for the Cartesian product.

The following example illustrates a nested loop with an implicit Cartesian product. If the inner table of a nested loop operation is not driven from the outer table, but from an independent row source, then the rows accessed can be the same as in a Cartesian product. Because the join condition is present but is applied after accessing the table, it is not a Cartesian product. However, the cost of accessing the table (rows accessed) is about the same.

```
SELECT a.attribute4,b.full_name
  FROM per_all_people_f b, so_lines_all a
 WHERE a.header_id = :b1
   AND b.employee_number = a.created_by
   AND b.business_group_id = :b2;

Plan
-------------------------------------------------
SELECT STATEMENT
 NESTED LOOPS
  TABLE ACCESS BY INDEX ROWID PER_ALL_PEOPLE_F
   INDEX RANGE SCAN PER_PEOPLE_F_FK1
  TABLE ACCESS BY INDEX ROWID SO_LINES_ALL
   INDEX RANGE SCAN SO_LINES_N1
```

In examining the statement, do the following:

1. Access `so_lines_all` using `header_id` (`so_line_n1`).
2. Join to `per_all_people_f` using `employee_number` (`per_peopl_f_n51`) for every row returned from `so_lines_all`.
3. Apply the `business_group_id` filter to `per_people_f`.

However, the execution plan shows something quite different. The datatype for `employee_number` is `VARCHAR2`, while `created_by` is `NUMBER`, so there is an implicit type conversion on `employee_number` disabling the index. The result is two independent sources:

- All rows from `per_people_f` for the `business_group_id`
- All rows from `so_lines` for the `header_id`

For every row in #1, every row in #2 is accessed, but returns only those rows that meet the following condition:

```
TO_NUMBER(b.employee_number) = a.created_by
```

Because there is a join condition, it does not show up as a Cartesian product. However, the number of data accesses are the same.

When the Optimizer Uses Cartesian Joins

The optimizer uses Cartesian joins when it is asked to join two tables with no join conditions.

Cartesian Join Hints

The hint ORDERED can cause a Cartesian join. By specifying a table before its join table is specified, the optimizer does a Cartesian join.

Example before using hint:

```
SELECT h.purchase_order_num, sum(l.revenue_amount)
  FROM ra_customers c, so_lines_all l, so_headers_all h
 WHERE c.customer_name = :b1
   AND h.customer_id = c.customer_id
   AND h.header_id = l.header_id
GROUP BY h.purchase_order_num;

Plan
-----------------------------------------------
SELECT STATEMENT
 SORT GROUP BY
  NESTED LOOPS
   NESTED LOOPS
    TABLE ACCESS BY INDEX ROWID RA_CUSTOMERS
     INDEX RANGE SCAN RA_CUSTOMERS_N1
    TABLE ACCESS BY INDEX ROWID SO_HEADERS_ALL
     INDEX RANGE SCAN SO_HEADERS_N1
   TABLE ACCESS BY INDEX ROWID SO_LINES_ALL
    INDEX RANGE SCAN SO_LINES_N1
```

The following is necessary:

1. Access customers using `customer_name`.
2. Join to `headers` using `customer_id`.
3. Join to `lines` using `line_id`.

This is exactly what the optimizer is doing.

Example after using hint:

```
SELECT /*+ORDERED */ h.purchase_order_num, sum(l.revenue_amount)
  FROM ra_customers c, so_lines_all l, so_headers_all h
 WHERE c.customer_name = :b1
   AND h.customer_id = c.customer_id
   AND h.header_id = l.header_id
GROUP BY h.purchase_order_num;

Plan
-----------------------------------------------
SELECT STATEMENT
 SORT GROUP BY
  MERGE JOIN
   SORT JOIN
    MERGE JOIN CARTESIAN
     TABLE ACCESS BY INDEX ROWID RA_CUSTOMERS
      INDEX RANGE SCAN RA_CUSTOMERS_N1
     SORT JOIN
      TABLE ACCESS FULL SO_LINES_ALL
   SORT JOIN
    TABLE ACCESS FULL SO_HEADERS_ALL
```

The optimizer now does the following:

1. Accesses `customers` using `customer_name`
2. Does a Cartesian product with `lines`, because it is the next column in the `FROM` clause
3. The rows from `customers` get multiplied several millionfold rows in `lines`
4. Sorts the resulting row set by (`customer_id`, `header_id`)
5. Does a merge join with `headers` sorted by (`customer_id`, `header_id`)

With a nonselective filter condition between `lines` and `customers`

```
SELECT /*+ORDERED */ h.purchase_order_num, SUM(l.revenue_amount)
  FROM ra_customers c, so_lines_all l, so_headers_all h
 WHERE c.customer_name = :b1
   AND h.customer_id = c.customer_id
   AND h.header_id = l.header_id
   AND c.price_list_id = l.price_list_id
GROUP BY h.purchase_order_num;
```

```
Plan
-----------------------------------------------
SELECT STATEMENT
SORT GROUP BY
  MERGE JOIN
   SORT JOIN
    NESTED LOOPS
     TABLE ACCESS BY INDEX ROWID RA_CUSTOMERS
      INDEX RANGE SCAN RA_CUSTOMERS_N1
     TABLE ACCESS FULL SO_LINES_ALL
   SORT JOIN
    TABLE ACCESS FULL SO_HEADERS_ALL
```

The execution plan is similar, but now, because the optimizer finds a join condition between the two tables, it removes the Cartesian. Even though an actual Cartesian product is not built, the plan is equally bad in terms of block accesses, if not worse. This is because nested loop joins do not perform well with large data sets.

Full Outer Joins

A full outer join acts like a combination of the left and right outer joins. In addition to the inner join, rows from both tables that have not been returned in the result of the inner join are preserved and extended with nulls.

In other words, full outer joins let you join tables together, yet still show rows which do not have corresponding rows in tables joined-to.

Full Outer Join Example

The following example shows all departments and all employees in each department, but also includes:

- Any employees without departments
- Any departments without employees

```
SELECT d.department_id, e.employee_id
  FROM employees e
  FULL OUTER JOIN departments d
    ON e.department_id = d.department_id
 ORDER BY d.department_id;
```

This is the output:

```
DEPARTMENT_ID EMPLOYEE_ID
------------- -----------
           10      255200
           20      255202
           20      255201
           30
           40      255203
                   255199
                   255204
                   255206
                   255205
```

How the CBO Chooses Execution Plans for Joins

Note: 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.

With the CBO, the optimizer generates a set of execution plans based on the possible join orders, join methods, 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.

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. Also, you can override the optimizer's choice of join method with hints.

> **See Also:** Chapter 5, "Optimizer Hints" for more information about optimizer hints

How the CBO Executes Anti-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 MERGE_AJ, HASH_AJ, or NL_AJ hint is used and various required conditions are met, which allow the transformation of the NOT IN uncorrelated subquery into a sort-merge or hash anti-join.

How the CBO Executes Semi-Joins

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 by default for `IN` or `EXISTS` subqueries that cannot be merged with the containing query, unless the `MERGE_SJ`, `HASH_SJ`, or `NL_SJ` hint is used and various required conditions are met, which allow the transformation of the subquery into a sort-merge or hash semi-join.

> **See Also:** Chapter 5, "Optimizer Hints" for information about optimizer hints

How the CBO Executes Star Queries

Some data warehouses are designed around a star schema, which includes a large fact table and several small dimension (lookup) tables. The fact table stores primary information. Each dimension table stores information about an 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, typically contains 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:** *Oracle9i Data Warehousing Guide* for more information about tuning star queries

Cost-Based Optimizer Parameters

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

OPTIMIZER_FEATURES_ENABLE Parameter

The OPTIMIZER_FEATURES_ENABLE parameter acts as an umbrella parameter for the CBO. This parameter can be used to enable a series of CBO-related features depending on the release. It accepts one of a list of valid string values corresponding to the release numbers, such as 8.0.4, 8.1.5, and so on.

The following statement enables the use of the optimizer features in Oracle8*i* release 8.1.6.

```
OPTIMIZER_FEATURES_ENABLE=8.1.6;
```

The above statement causes the optimizer features available in release 8.1.6 to be used in generating query plans. For example, one of the changes in release 8.1.6 is the use of ALL_ROWS or FIRST_ROWS optimizer mode for recursive user SQL generated by PL/SQL procedures. Prior to Oracle8*i* release 8.1.6, only RULE or CHOOSE optimizer mode was used for such recursive SQL, and when the user explicitly set the OPTIMIZER_MODE parameter to FIRST_ROWS or ALL_ROWS, a CHOOSE mode was used instead.

The OPTIMIZER_FEATURES_ENABLE parameter was introduced in Oracle8 release 8.0.4. The main goal was to allow customers the ability to upgrade the Oracle server, yet preserve the old behavior of the CBO after the upgrade. For example, when you upgrade the Oracle server from release 8.1.5 to release 8.1.6, the default value of the OPTIMIZER_FEATURES_ENABLE parameter changes from 8.1.5 to 8.1.6. This results in the cost--based optimizer enabling optimization features based on 8.1.6, as opposed to 8.1.5. For plan stability or backward compatibility reasons, you might not want the query plans to change based on the new optimizer features of release 8.1.6. In such a case, set the OPTIMIZER_FEATURES_ENABLE parameter to an earlier version, such as 8.1.5. To disable all new optimizer features in release 8.1.6, set the parameter as follows:

```
OPTIMIZER_FEATURES_ENABLE=8.1.5;
```

To preserve an even older behavior of the CBO, such as release 8.0.4, set the parameter as follows:

```
OPTIMIZER_FEATURES_ENABLE=8.0.4;
```

The above statement disables all new optimizer features that were added in releases following 8.0.4.

> **Note:** If you upgrade to a new release and you want to enable the features with that release, then you do not need to explicitly set the OPTIMIZER_FEATURES_ENABLE parameter.

Oracle Corporation does not recommend explicitly setting the OPTIMIZER_FEATURES_ENABLE parameter to an earlier release. Instead, execution plan or query performance issues should be resolved on a case by case basis.

The table below describes some of the optimizer features that are enabled when you set the OPTIMIZER_FEATURES_ENABLE parameter to each of the following release values.

Table 1–1 Features Included with the OPTIMIZER_FEATURES_ENABLE Parameter

Set to Value	New Features include
8.0.4	Index fast full scan
	Ordered nested loop join method
8.0.5	No new features
8.0.6	Improved outer join cardinality estimation
8.1.4	No new features
8.1.5	Improved verification of NULLs inclusion in B-tree indexes
8.1.6	Use of FIRST_ROWS or ALL_ROWS mode for user recursive SQL
	Use random distribution of left input of nested loop join
	Improved row length calculation
	Improved method of computing selectivity based on histogram
	Partition pruning based on predicates in a subquery
8.1.7	Common subexpression optimization
	Use statistics of a column imbedded in some selected functions such as TO_CHAR to compute selectivity
	Improved partition statistics aggregation

Table 1–1 Features Included with the OPTIMIZER_FEATURES_ENABLE Parameter

Set to Value	New Features include
9.0.1	Peeking of user-defined bind variables
	Complex view merging
	Push-join predicate
	Consideration of bitmap access paths for tables with only B-tree indexes
	Subquery unnesting
	Index joins

Peeking of User-Defined Bind Variables

The CBO peeks at the values of user-defined bind variables on the first invocation of a cursor. This lets the optimizer determine the selectivity of any `WHERE` clause condition, as well as if literals had been used instead of bind variables. On subsequent invocations of the cursor, no peeking takes place, and the cursor is shared based on the standard cursor-sharing criteria, even if subsequent invocations use different bind values.

When bind variables are used in a statement, it is assumed that cursor sharing is intended and that different invocations are supposed to use the same execution plan. If different invocations of the cursor would significantly benefit from different execution plans, then bind variables may have been used inappropriately in the SQL statement.

Other CBO Parameters

The following table lists initialization parameters that can be used to control the behavior of the CBO. These parameters can be used to enable various optimizer features in order to improve the performance of SQL execution.

`CURSOR_SHARING` — This parameter converts literal values in SQL statements to bind variables. This improves cursor sharing and can affect the execution plans of SQL statements. The optimizer generates the execution plan based on the presence of the bind variables and not the actual literal values.

DB_FILE_MULTIBLOCK_READ_COUNT	This parameter specifies the number of blocks that are read in a single I/O during a full table scan or index fast full scan. The optimizer uses the value of DB_FILE_MULTIBLOCK_READ_COUNT to cost full table scans and index fast full scans. Larger values result in a cheaper cost for full table scans and can result in the optimizer choosing a full table scan over an index scan.
HASH_AREA_SIZE	This parameter specifies the amount of memory (in bytes) to be used for hash joins. The CBO uses this parameter to cost a hash join operation. Larger values for HASH_AREA_SIZE reduce the cost of hash joins.
HASH_JOIN_ENABLED	This parameter can be used to enable or disable the use of hash joins as a join method chosen by the optimizer. When set to true, the optimizer considers hash joins as a possible join method. The CBO chooses a hash join if the cost is better than the other join methods, such as nested loops or sort merge joins.
OPTIMIZER_INDEX_CACHING	This parameter is used to control the costing of an index probe in conjunction with a nested loop. The range of values 0 to 100 for OPTIMIZER_INDEX_CACHING control the degree of caching of the index buffers in the buffer cache. A value of 100 infers that 100% of the index blocks are likely to be found in the buffer cache; hence, the optimizer adjusts the cost of an index probe/nested loop accordingly. Be careful when using this parameter, because execution plans can change in favor of index caching.
OPTIMIZER_INDEX_COST_ADJ	This parameter can be used to adjust the cost of index probes. The range of values is [1..10000]. The default value is 100, which means that indexes are evaluated as an access path based on the normal costing model. A value of 10 means that the cost of an index access path is 1/10th the normal cost of an index access path.

OPTIMIZER_MAX_PERMUTATIONS	This parameter controls the maximum number of permutations that the CBO considers when generating execution plans for SQL statements with joins. Values less than the default of 80,000 result in the optimizer considering more starting tables. The range of values is [4..262140]. This parameter can be used to reduce parse times for complex SQL statements that join a large number of tables. However, reducing it's value can result in the optimizer missing an optimal join permutation.
OPTIMIZER_MODE OPTIMIZER_GOAL	These initialization parameters set the mode of the optimizer at instance startup. The possible values are RULE, CHOOSE, ALL_ROWS, FIRST_ROWS_*n*, and FIRST_ROWS. A value of ALL_ROWS is the default CBO behavior when statistics are present (that is, optimize for the best overall throughput). A value of FIRST_ROWS_*n* or FIRST_ROWS causes the first rows optimization method to be used by the CBO. The first rows optimization method optimizes for best response time, such that the first set of rows is returned quickly. The *n* factor controls the number of the first set of rows to be returned. The CBO optimizes the plan based on the first rows method in conjunction with the number of rows specified by the value of *n*. A value of RULE causes the rule-based optimizer (RBO) to be used to generate executions plans. CHOOSE causes the cost-based optimizer to be used if statistics are present; otherwise, the RBO is used.

PARALLEL_BROADCAST_ENABLED	This parameter controls the behavior of parallel execution for SQL statements that use parallel query and involve joins to small reference or lookup tables. If set to `true`, then the rows of the small table are broadcast to each slave. This improves the performance of parallel execution for queries that perform hash joins or merge joins between a large table and a small lookup or reference table.
PARTITION_VIEW_ENABLED	This parameter enables the partition view pruning feature. If set to `true`, then the CBO scans only the required partitions based on the view predicates / filters.
QUERY_REWRITE_ENABLED	This parameter enables the query rewrite feature, which works in conjunction with materialized views. If set to `true`, then the CBO considers query rewrites using materialized views to satisfy the original query. This parameter also controls whether or not function based indexes are used.
SORT_AREA_SIZE	This parameter specifies the amount of memory (in bytes) that will be used to perform sorts. If a sort operation is performed, and if the amount of data to be sorted exceeds the value of SORT_AREA_SIZE, then the amount of data above the value of SORT_AREA_SIZE is written to the temporary tablespace. The CBO uses the value of SORT_AREA_SIZE to cost sort operations including sort merge joins. Larger values for SORT_AREA_SIZE result in cheaper CBO costs for sort operations.
STAR_TRANSFORMATION_ENABLED	This parameter, if set to `true`, enables the CBO to cost a star transformation for star queries. The star transformation combines the bitmap indexes on the various fact table columns rather than using a Cartesian approach.

See Also: *Oracle9i Database Reference* for complete information about each parameter

Overview of the 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 standalone functions.
- Associate selectivity function and default selectivity with methods of object types, package functions, and standalone functions.
- Associate statistics collection functions with domain indexes and columns of tables.
- Order predicates with functions based on cost.
- Select a user-defined access path (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:** *Oracle9i Data Cartridge Developer's Guide* for details about the extensible optimizer

Understanding 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.

Understanding User-Defined Selectivity

The selectivity of a predicate in a SQL statement is used to estimate the cost of a particular access path; 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, standalone 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`.

Understanding 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 might 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 standalone 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.

2

Optimizer Operations

This chapter expands on the ideas introduced in Chapter 1 and explains optimizer actions in greater detail for specific cases. This chapter describes how the cost-based optimizer evaluates expressions and performs specific operations. It also explains how the CBO transforms some SQL statements into others to achieve the same goal more efficiently.

This chapter contains the following sections:

- How the Optimizer Performs Operations
- How the Optimizer Transforms SQL Statements

How the Optimizer Performs Operations

The optimizer fully evaluates expressions whenever possible and translates certain syntactic constructs into equivalent constructs. The reason for this is that either Oracle can more quickly evaluate the resulting expression than the original expression, or the original expression is merely a syntactic equivalent of the resulting expression. Sometimes, different SQL constructs can 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:

- How the CBO Evaluates IN-List Iterators
- How the CBO Evaluates Concatenation
- How the CBO Evaluates Remote Operations
- How the CBO Executes Distributed Statements
- How the CBO Executes Sort Operations
- How the CBO Executes Views
- How the CBO Evaluates Constants
- How the CBO Evaluates the UNION / UNION ALL Operators
- How the CBO Evaluates the LIKE Operator
- How the CBO Evaluates the IN Operator
- How the CBO Evaluates the ANY or SOME Operator
- How the CBO Evaluates the ALL Operator
- How the CBO Evaluates the BETWEEN Operator
- How the CBO Evaluates the NOT Operator
- How the CBO Evaluates Transitivity
- How the CBO Optimizes Common Subexpressions
- How the CBO Evaluates DETERMINISTIC Functions

How the CBO Evaluates IN-List Iterators

This operator is used when there is an IN clause with values.

The execution plan is identical to what would result for a statement with an equality clause instead of IN. There is one additional step: the IN-list iterator that feeds the equality clause with unique values from the IN-list.

IN-List Iterator Examples

Both the statements below are equivalent and produce the same plan:

```
SELECT header_id, line_id, revenue_amount
  FROM so_lines_all
 WHERE header_id in (1011,1012,1013);

SELECT header_id, line_id, revenue_amount
  FROM so_lines_all
 WHERE header_id = 1011
    OR header_id = 1012
    OR header_id = 1013;

Plan
-------------------------------------------------
SELECT STATEMENT
 INLIST ITERATOR
  TABLE ACCESS BY INDEX ROWID SO_LINES_ALL
   INDEX RANGE SCAN SO_LINES_N1
```

This is equivalent to the following statement, with :b1 being bound to the different unique values by the IN-list iterator:

```
SELECT header_id, line_id, revenue_amount
  FROM so_lines_all l
 WHERE header_id = :b1;

Plan
-------------------------------------------------
SELECT STATEMENT
 TABLE ACCESS BY INDEX ROWID SO_LINES_ALL
  INDEX RANGE SCAN SO_LINES_N1
```

The following example uses a unique index. Because there is a sort involved on the IN-list, even with complete keys of unique indexes, there is still a range scan.

```
SELECT header_id, line_id, revenue_amount
  FROM so_lines_all
 WHERE line_id IN (1011,1012,1013);
```

```
Plan
-----------------------------------------------
SELECT STATEMENT
 INLIST ITERATOR
  TABLE ACCESS BY INDEX ROWID SO_LINES_ALL
   INDEX RANGE SCAN SO_LINES_U1
```

In the following example, this operator can be used when driving into a table with a nested loop operation.

```
SELECT h.header_id, l.line_id, l.revenue_amount
  FROM so_headers_all h, so_lines_all l
 WHERE l.inventory_item_id = :b1
   AND h.order_number = l.header_id
   AND h.order_type_id IN (1,2,3);

Plan
-----------------------------------------------
SELECT STATEMENT
 NESTED LOOPS
  TABLE ACCESS BY INDEX ROWID SO_LINES_ALL
   INDEX RANGE SCAN SO_LINES_N5
  INLIST ITERATOR
   TABLE ACCESS BY INDEX ROWID SO_HEADERS_ALL
    INDEX RANGE SCAN SO_HEADERS_U2
```

In the example below, this operator is especially useful if there is an expensive first step that you do not want to repeat for every `IN`-list element. In the example below, even though there are three `IN`-list elements, the full scan on `so_lines_all` happens only once.

```
SELECT h.header_id, l.line_id, l.revenue_amount
  FROM so_headers_all h, so_lines_all l
 WHERE l.s7 = :b1
   AND h.order_number = l.header_id
   AND h.order_type_id IN (1,2,3);

Plan
-----------------------------------------------
SELECT STATEMENT
 NESTED LOOPS
  TABLE ACCESS FULL SO_LINES_ALL
  INLIST ITERATOR
   TABLE ACCESS BY INDEX ROWID SO_HEADERS_ALL
    INDEX RANGE SCAN SO_HEADERS_U2
```

When the Optimizer Uses IN-List Iterators

The optimizer uses an IN-list iterator when an IN clause is specified with values, and the optimizer finds a selective index for that column. If there are multiple OR clauses using the same index, then the optimizer chooses this operation rather than CONCATENATION or UNION ALL, because it is more efficient.

IN-List Iterator Hints

There are no hints for this operation. You can provide a hint to use the index in question, which can cause this operation.

Example before using hint:

```
SELECT h.customer_id, l.line_id, l.revenue_amount
  FROM so_lines_all l, so_headers_all h
 WHERE l.s7 = 20
   AND h.original_system_reference = l.attribute5
   AND h.original_system_source_code IN (1013,1014);

Plan
-------------------------------------------------
SELECT STATEMENT
 NESTED LOOPS
  TABLE ACCESS FULL SO_LINES_ALL
  TABLE ACCESS BY INDEX ROWID SO_HEADERS_ALL
   INDEX RANGE SCAN SO_HEADERS_N5
```

Example after using hint:

```
SELECT /*+INDEX(h so_headers_n9 */ h.customer_id, l.line_id, l.revenue_amount
  FROM so_lines_all l, so_headers_all h
 WHERE l.s7 = 20
   AND h.original_system_reference = l.attribute5
   AND h.original_system_source_code IN (1013,1014);

Plan
-------------------------------------------------
SELECT STATEMENT
 NESTED LOOPS
  TABLE ACCESS FULL SO_LINES_ALL
  INLIST ITERATOR
   TABLE ACCESS BY INDEX ROWID SO_HEADERS_ALL
    INDEX RANGE SCAN SO_HEADERS_N9
```

How the CBO Evaluates Concatenation

Concatenation is useful for statements with different conditions combined with an OR clause.

Concatenation Examples

Below, there are two plans, each accessing the table via the appropriate index, combined via concatenation.

```
SELECT l.header_id, l.line_id, l.revenue_amount
  FROM so_lines_all l
 WHERE l.parent_line_id = :b1
    OR l.service_parent_line_id = :b1;

Plan
-----------------------------------------------
SELECT STATEMENT
 CONCATENATION
  TABLE ACCESS BY INDEX ROWID SO_LINES_ALL
   INDEX RANGE SCAN SO_LINES_N20
  TABLE ACCESS BY INDEX ROWID SO_LINES_ALL
   INDEX RANGE SCAN SO_LINES_N17
```

The plan does not return duplicate rows, so for each component it appends a negation of the previous components. In this case, the components will be the following:

- `l.parent_line_id = :b1`
- `l.service_parent_line_id = :b1 and l.parent_line_id != :b1`

The example below shows that concatenation is particularly useful in optimizing queries with OR conditions. With concatenation, you get a good execution plan with appropriate indexes.

```
SELECT p.header_id, l.line_id, l.revenue_amount
  FROM so_lines_all p , so_lines_all l
 WHERE p.header_id = :b1
   AND (l.parent_line_id = p.line_id
       OR l.service_parent_line_id = p.line_id);
```

```
Plan
-----------------------------------------------
SELECT STATEMENT
 CONCATENATION
  NESTED LOOPS
   TABLE ACCESS BY INDEX ROWID SO_LINES_ALL
    INDEX RANGE SCAN SO_LINES_N1
   TABLE ACCESS BY INDEX ROWID SO_LINES_ALL
    INDEX RANGE SCAN SO_LINES_N20
  NESTED LOOPS
   TABLE ACCESS BY INDEX ROWID SO_LINES_ALL
    INDEX RANGE SCAN SO_LINES_N1
   TABLE ACCESS BY INDEX ROWID SO_LINES_ALL
    INDEX RANGE SCAN SO_LINES_N17
```

Trying to execute the statement in a single query (by using a hint to disable concatenation) produces a poor execution plan. Because the optimizer has two paths to follow and has been instructed not to decompose the query, it needs to access all the rows in the second table to see if any match one of the conditions. For example:

```
SELECT /*+NO_EXPAND */ p.header_id, l.line_id, l.revenue_amount
  FROM so_lines_all p, so_lines_all l
 WHERE p.header_id = :b1
   AND (l.parent_line_id = p.line_id
       OR l.service_parent_line_id = p.line_id);

Plan
-----------------------------------------------
SELECT STATEMENT
 NESTED LOOPS
  TABLE ACCESS BY INDEX ROWID SO_LINES_ALL
   INDEX RANGE SCAN SO_LINES_N1
  TABLE ACCESS FULL SO_LINES_ALL
```

Concatenation Hints

Use the hint USE_CONCAT for this operation.

When not to use concatenation:

1. OR conditions are on same column can use the IN-list operator, which is a more efficient than concatenation.
2. If there is an expensive step that gets repeated for every concatenation

In the example below, there is a full scan on `so_lines_all` as the first stage. The optimizer chooses to use a single column index for the second table, but we want it to use a two column index.

Hints can force concatenation, but this repeats the initial full scan, which is not desirable. Instead, if you provide a hint to use the two column index, then the optimizer switches to that with an `IN`-list operator. The initial scan is not repeated, and there is a better execution plan.

Initial Statement

```
SELECT h.customer_id, l.line_id, l.revenue_amount
  FROM so_lines_all l, so_headers_all h
 WHERE l.s7 = 20
   AND h.original_system_reference = l.attribute5
   AND h.original_system_source_code IN (1013,1014);

Plan
-------------------------------------------------
SELECT STATEMENT
 NESTED LOOPS
  TABLE ACCESS FULL SO_LINES_ALL
  TABLE ACCESS BY INDEX ROWID SO_HEADERS_ALL
   INDEX RANGE SCAN SO_HEADERS_N5
```

Example using concatenation hint:

```
SELECT h.customer_id, l.line_id, l.revenue_amount
  FROM so_lines_all l, so_headers_all h
 WHERE l.s7 = 20
   AND h.original_system_reference = l.attribute5
   AND h.original_system_source_code IN (1013,1014);

Plan
-------------------------------------------------
SELECT STATEMENT
 CONCATENATION
  NESTED LOOPS
   TABLE ACCESS FULL SO_LINES_ALL
   TABLE ACCESS BY INDEX ROWID SO_HEADERS_ALL
    INDEX RANGE SCAN SO_HEADERS_N9
  NESTED LOOPS
   TABLE ACCESS FULL SO_LINES_ALL
   TABLE ACCESS BY INDEX ROWID SO_HEADERS_ALL
    INDEX RANGE SCAN SO_HEADERS_N9
```

Example using index hint:

```
SELECT h.customer_id, l.line_id, l.revenue_amount
  FROM so_lines_all l, so_headers_all h
 WHERE l.s7 = 20
   AND h.original_system_reference = l.attribute5
   AND h.original_system_source_code IN (1013,1014);

Plan
-------------------------------------------------
SELECT STATEMENT
 NESTED LOOPS
  TABLE ACCESS FULL SO_LINES_ALL
  INLIST ITERATOR
   TABLE ACCESS BY INDEX ROWID SO_HEADERS_ALL
    INDEX RANGE SCAN SO_HEADERS_N9
```

How the CBO Evaluates Remote Operations

The remote operation indicates that there is a table from another database being accessed via a database link.

Remote Examples

The following example has a remote driving table:

```
SELECT c.customer_name, count(*)
  FROM ra_customers c, so_headers_all@oe h
 WHERE c.customer_id = h.customer_id
   AND h.order_number = :b1
GROUP BY c.customer_name;

Plan
-------------------------------------------------
SELECT STATEMENT
 SORT GROUP BY
  NESTED LOOPS
   REMOTE
   TABLE ACCESS BY INDEX ROWID RA_CUSTOMERS
    INDEX UNIQUE SCAN RA_CUSTOMERS_U1
```

A remote database query obtained from the library cache:

```
SELECT "ORDER_NUMBER","CUSTOMER_ID"
  FROM "SO_HEADERS_ALL" "H"
 WHERE "ORDER_NUMBER"=:"SYS_B_0";
```

The example below has a local driving table. Several factors are influencing the execution plan:

- Network round-trips can be several orders of magnitude more expensive than the physical and logical I/Os.
- The optimizer does not have any statistics on the remote database, which might not be an Oracle database.

In general, the optimizer chooses to access the remote tables first, before accessing the local tables. This works well for cases like the previous example, where the driving table was the remote table. However, if the driving table is the local table, then there might not be any selective way of accessing the remote table without first accessing the local tables. In such cases, you might need to provide appropriate hints to avoid performance problems.

Original Query:

```
SELECT c.customer_name, h.order_number
  FROM ra_customers c, so_headers_all@oe h
 WHERE c.customer_id = h.customer_id
   AND c.customer_name LIKE :b1;

Plan
-------------------------------------------------
SELECT STATEMENT
 MERGE JOIN
  REMOTE
  SORT JOIN
   TABLE ACCESS BY INDEX ROWID RA_CUSTOMERS
    INDEX RANGE SCAN RA_CUSTOMERS_N1
```

Remote database query obtained from the library cache:

```
SELECT "ORDER_NUMBER","CUSTOMER_ID"
  FROM "SO_HEADERS_ALL" "H"
 WHERE "CUSTOMER_ID" IS NOT NULL
ORDER BY "CUSTOMER_ID";
```

After hints:

```
SELECT /*+USE_NL(c h) */ c.customer_name, h.order_number
  FROM ra_customers c, so_headers_all@oe h
 WHERE c.customer_id = h.customer_id
   AND c.customer_name LIKE :b1;
```

```
Plan
-------------------------------------------------
SELECT STATEMENT
 NESTED LOOPS
  TABLE ACCESS BY INDEX ROWID RA_CUSTOMERS
   INDEX RANGE SCAN RA_CUSTOMERS_N1
  REMOTE
```

Remote database query obtained from the library cache:

```
SELECT /*+ USE_NL("H") */ "ORDER_NUMBER","CUSTOMER_ID"
  FROM "SO_HEADERS_ALL" "H" WHERE :1="CUSTOMER_ID"
FILTER;
```

This indicates that the optimizer is applying a filter condition to filter out rows, which could not be applied when the table was accessed.

The example below uses no filter. Besides the conditions used in the access path, a table might have additional conditions to filter rows when the table is visited. Conditions that get applied when the table is accessed, like `attribute1 IS NOT NULL`, do not show up as `FILTER`.

```
SELECT h.order_number
  FROM so_headers_all h
 WHERE h.open_flag = 'Y'
   AND attribute1 IS NOT NULL;

Plan
-------------------------------------------------
SELECT STATEMENT
 TABLE ACCESS BY INDEX ROWID SO_HEADERS_ALL
  INDEX RANGE SCAN SO_HEADERS_N2
```

Example - filter due to `GROUP BY` condition:

```
SELECT h.order_number, count(*)
  FROM so_headers_all h
 WHERE h.open_flag = 'Y'
   AND attribute1 IS NOT NULL
GROUP BY h.order_number
HAVING COUNT(*) = 1 ß Filter condition;
```

```
Plan
-------------------------------------------------
SELECT STATEMENT
 FILTER
  SORT GROUP BY
   TABLE ACCESS BY INDEX ROWID SO_HEADERS_ALL
    INDEX RANGE SCAN SO_HEADERS_N2
```

Example - filter due to a subquery:

In the example below, for every row meeting the condition of the outer query, the correlated `EXISTS` subquery is executed. If a row meeting the condition is found in the `so_lines_all` table, then the row from `so_headers_all` is returned.

```
SELECT h.order_number
  FROM so_headers_all h
 WHERE h.open_flag = 'Y'
   AND EXISTS (SELECT null FROM so_lines_all l
               WHERE l.header_id = h.header_id
               AND l.revenue_amount > 10000);

Plan
-------------------------------------------------
SELECT STATEMENT
 FILTER
  TABLE ACCESS BY INDEX ROWID SO_HEADERS_ALL
   INDEX RANGE SCAN SO_HEADERS_N2
  TABLE ACCESS BY INDEX ROWID SO_LINES_ALL
   INDEX RANGE SCAN SO_LINES_N1
```

How the CBO Executes 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 can 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:** Chapter 6, "Optimizing SQL Statements" for more information on tuning distributed queries

How the CBO Executes Sort Operations

Sort operations happen when users specify some operation that requires a sort. Commonly encountered operations include the following:

- Sort Unique
- Sort Aggregate
- Sort Group By
- Sort Join
- Sort Order By

Sort Unique

This operation occurs if a user specifies a `DISTINCT` clause, or if there is an operation requiring unique values for the next step.

Example - Distinct clause causing SORT UNIQUE

```
SELECT DISTINCT last_name, first_name
 FROM per_all_people_f
 WHERE full_name LIKE :b1;
```

```
Plan
-------------------------------------------------
SELECT STATEMENT
 SORT UNIQUE
  TABLE ACCESS BY INDEX ROWID PER_ALL_PEOPLE_F
   INDEX RANGE SCAN PER_PEOPLE_F_N54
```

Example - IN Subquery causing SORT UNIQUE Sort unique is happening to provide the outer query with a unique list of `header_id`. There is a view that indicates that the `IN` subquery has been un-nested by transforming into a view.

```
SELECT c.customer_name, h.order_number
  FROM ra_customers c, so_headers_all h
 WHERE c.customer_id = h.customer_id
   AND h.header_id in
     (SELECT l.header_id FROM so_lines_all l
      WHERE l.inventory_item_id = :b1
      AND ordered_quantity > 10);

Plan
-------------------------------------------------
SELECT STATEMENT
 NESTED LOOPS
  NESTED LOOPS
   VIEW  VW_NSO_1
    SORT UNIQUE
     TABLE ACCESS BY INDEX ROWID SO_LINES_ALL
      INDEX RANGE SCAN SO_LINES_N5
   TABLE ACCESS BY INDEX ROWID SO_HEADERS_ALL
    INDEX UNIQUE SCAN SO_HEADERS_U1
  TABLE ACCESS BY INDEX ROWID RA_CUSTOMERS
   INDEX UNIQUE SCAN RA_CUSTOMERS_U1
```

Example - IN Subquery not causing SORT UNIQUE If the optimizer can guarantee (with unique keys) that duplicate values will not be passed, then a sort can be avoided.

```
UPDATE so_lines_all l
   SET line_status = 'HOLD'
 WHERE l.header_id IN
     ( SELECT h.header_id FROM so_headers_all h
        WHERE h.customer_id = :b1);
```

```
Plan
-------------------------------------------------
UPDATE STATEMENT
 UPDATE  SO_LINES_ALL
  NESTED LOOPS
   TABLE ACCESS BY INDEX ROWID SO_HEADERS_ALL
    INDEX RANGE SCAN SO_HEADERS_N1
   INDEX RANGE SCAN SO_LINES_N1
```

Sort Aggregate

This operation does not actually involve a sort. It is used when aggregates are being computed across the whole set of rows.

```
SELECT SUM(l.revenue_amount)
  FROM so_lines_all l
 WHERE l.header_id = :b1;

Plan
-------------------------------------------------
SELECT STATEMENT
 SORT AGGREGATE
  TABLE ACCESS BY INDEX ROWID SO_LINES_ALL
   INDEX RANGE SCAN SO_LINES_N1
```

Sort Group By

This is used when aggregates are being computed for different groups in the data. The sort is required to separate the rows in different groups.

```
SELECT created_by, SUM(l.revenue_amount)
  FROM so_lines_all l
 WHERE header_id > :b1
GROUP BY created_by;

Plan
-------------------------------------------------
SELECT STATEMENT
 SORT GROUP BY
  TABLE ACCESS BY INDEX ROWID SO_LINES_ALL
   INDEX RANGE SCAN SO_LINES_N1
```

Sort Join

This happens during sort merge joins if the rows need to be sorted by the join key.

```
SELECT SUM(l.revenue_amount), l2.creation_date
  FROM so_lines_all l, so_lines_All l2
 WHERE l.creation_date < l2.creation_date
   AND l.header_id <> l2.header_id
GROUP BY l2.creation_date, l2.line_id;

Plan
-------------------------------------------------
SELECT STATEMENT
 SORT GROUP BY
  MERGE JOIN
   SORT JOIN
    TABLE ACCESS FULL SO_LINES_ALL
   FILTER
    SORT JOIN
     TABLE ACCESS FULL SO_LINES_ALL
```

Sort Order By

This operation is required when the statement specifies an ORDER BY that cannot be satisfied by one of the indexes.

```
SELECT h.order_number
  FROM so_headers_all h
 WHERE h.customer_id = :b1
 ORDER BY h.creation_date DESC;

Plan
-------------------------------------------------
SELECT STATEMENT
 SORT ORDER BY
  TABLE ACCESS BY INDEX ROWID SO_HEADERS_ALL
   INDEX RANGE SCAN SO_HEADERS_N1
```

How the CBO Executes Views

Either of the following could cause the view:

- A complex view has not been decomposed.
- A temporary/inline view is being used.

Example - Using a view

```
SELECT order_number
FROM orders
 WHERE customer_id = :b1
   AND revenue > :b2;

Plan
-------------------------------------------------
SELECT STATEMENT
 VIEW  ORDERS
  FILTER
   SORT GROUP BY
    NESTED LOOPS
     TABLE ACCESS BY INDEX ROWID SO_HEADERS_ALL
      INDEX RANGE SCAN SO_HEADERS_N1
     TABLE ACCESS BY INDEX ROWID SO_LINES_ALL
      INDEX RANGE SCAN SO_LINES_N1
```

View Definition

There is a view due to the `IN` subquery requiring a sort unique on the values being selected. This view would be unnecessary if the columns being selected were unique, not requiring a sort.

```
SELECT c.customer_name, h.order_number
  FROM ra_customers c, so_headers_all h
 WHERE c.customer_id = h.customer_id
   AND h.header_id IN
     (SELECT l.header_id FROM so_lines_all l
      WHERE l.inventory_item_id = :b1
      AND ordered_quantity > 10);
Plan
-------------------------------------------------
SELECT STATEMENT
 NESTED LOOPS
  NESTED LOOPS
   VIEW  VW_NSO_1
    SORT UNIQUE
     TABLE ACCESS BY INDEX ROWID SO_LINES_ALL
      INDEX RANGE SCAN SO_LINES_N5
   TABLE ACCESS BY INDEX ROWID SO_HEADERS_ALL
    INDEX UNIQUE SCAN SO_HEADERS_U1
  TABLE ACCESS BY INDEX ROWID RA_CUSTOMERS
   INDEX UNIQUE SCAN RA_CUSTOMERS_U1
```

Example - VIEW due to an Inline Query

In the example below, the distribution of orders and revenue by the number of lines / order is examined. In order to do the double grouping, temporary inline views are used.

```
SELECT COUNT(*) "Orders", cnt "Lines", sum(rev) "Revenue"
FROM (SELECT header_id, COUNT(*) cnt, SUM(revenue_amount) rev
        FROM so_lines_all
       GROUP BY header_id)
GROUP BY cnt;

Plan
-----------------------------------------------
SELECT STATEMENT
 SORT GROUP BY
  VIEW
   SORT GROUP BY
    TABLE ACCESS FULL SO_LINES_ALL
```

How the CBO Evaluates 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 preceding examples, the optimizer does not simplify the third expression into the second. For this reason, application developers write conditions that compare columns with constants whenever possible, rather than conditions with expressions involving columns.

How the CBO Evaluates the UNION/UNION ALL Operators

This operator is useful for combining OR clauses into one compound statement or for breaking up a complex statement into a compound statement containing simpler select statements that are easier to optimize and understand.

Like concatenation, you do not want to duplicate expensive operations by using UNION ALL.

When the Optimizer Uses UNION/UNION ALL

The optimizer uses UNION/UNION ALL when the SQL statement contains UNION/UNION ALL clauses.

UNION/UNION ALL Examples

The following examples find customers who are new or have open orders.

Example without the UNION clause:

```
SELECT c.customer_name, c.creation_date
  FROM ra_customers c
 WHERE c.creation_date > SYSDATE - 30
    OR customer_id IN
     (SELECT customer_id FROM so_headers_all h
        WHERE h.open_flag = 'Y');

Plan
-------------------------------------------------
SELECT STATEMENT
 FILTER
  TABLE ACCESS FULL RA_CUSTOMERS
  TABLE ACCESS BY INDEX ROWID SO_HEADERS_ALL
   INDEX RANGE SCAN SO_HEADERS_N1
```

Because the driving conditions come from different tables, you cannot execute it effectively in a single statement.

With a UNION clause, you can break it up into two statements:

1. new customers

2. customers with open orders

These two statements can be optimized easily. Because you do not want duplicates (because some customers meet both criteria), use UNION rather than UNION ALL,

which eliminates duplicates by using a sort. If you were using two disjoint sets, then you could have used UNION ALL, eliminating the sort.

Example with the UNION clause:

```
SELECT c.customer_name, c.creation_date
  FROM ra_customers c
 WHERE c.creation_date > SYSDATE - 30
UNION ALL
SELECT c.customer_name, c.creation_date
  FROM ra_customers c
 WHERE customer_id IN
     (SELECT customer_id FROM so_headers_all h
        WHERE h.open_flag = 'Y');

Plan
-------------------------------------------------
SELECT STATEMENT
 SORT UNIQUE
  UNION-ALL
   TABLE ACCESS BY INDEX ROWID RA_CUSTOMERS
    INDEX RANGE SCAN RA_CUSTOMERS_N2
   NESTED LOOPS
    TABLE ACCESS BY INDEX ROWID SO_HEADERS_ALL
     INDEX RANGE SCAN SO_HEADERS_N2
    TABLE ACCESS BY INDEX ROWID RA_CUSTOMERS
     INDEX UNIQUE SCAN RA_CUSTOMERS_U1
```

UNION/UNION ALL Hints

There are no hints for this operation.

How the CBO Evaluates the 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.

How the CBO Evaluates the 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: "IN Subquery Example" on page 2-35

How the CBO Evaluates the 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)
```

How the CBO Evaluates the 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)
```

How the CBO Evaluates the 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
```

How the CBO Evaluates the 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
```

How the CBO Evaluates Transitivity

If two conditions in the WHERE clause involve a common column, then the optimizer sometimes can infer a third condition using the transitivity principle. The optimizer can then use the inferred condition to optimize the statement. The inferred condition potentially could 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.

For 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
```

How the CBO Optimizes Common Subexpressions

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

Common subexpression optimization is enabled with the initialization parameter OPTIMIZER_FEATURES_ENABLE.

A query is considered valid for common subexpression optimization 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 Optimization

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 previous query is transformed by common subexpression optimization to 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));
```

How the CBO Evaluates 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 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 globalization support parameters. Furthermore, if the function is redefined in the future, then its output return value must 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 to using a precalculated value instead of executing the function again.

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 might 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:

- *Oracle9i Application Developer's Guide - Fundamentals* for more information on DETERMINISTIC functions
- *Oracle9i SQL Reference* for descriptions of CREATE FUNCTION, CREATE INDEX, and CREATE MATERIALIZED VIEW
- *Oracle9i Database Concepts* for a description of function-based indexes
- *Oracle9i Data Warehousing Guide* for detailed information about materialized views

How the Optimizer Transforms SQL Statements

SQL is a very flexible query language; there are often many statements you could use to achieve the same goal. Sometimes, the optimizer (Query Transformer) 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:

- How the CBO Transforms ORs into Compound Queries
- How the CBO Unnests Subqueries
- How the CBO Merges Views
- How the CBO Pushes Predicates
- How the CBO Executes Compound Queries

> **See Also:** "Understanding Joins" on page 1-42 for additional information about optimizing statements that contain joins, semi-joins, or anti-joins

How the CBO Transforms 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 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 might 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 `OR`s on the same column; instead, it uses the `INLIST` iterator operator.

> **See Also:** "Understanding Access Paths for the RBO" on page 8-2 and "How the CBO Chooses an Access Path" on page 1-39 for information on access paths and how indexes make them available

For 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;
```

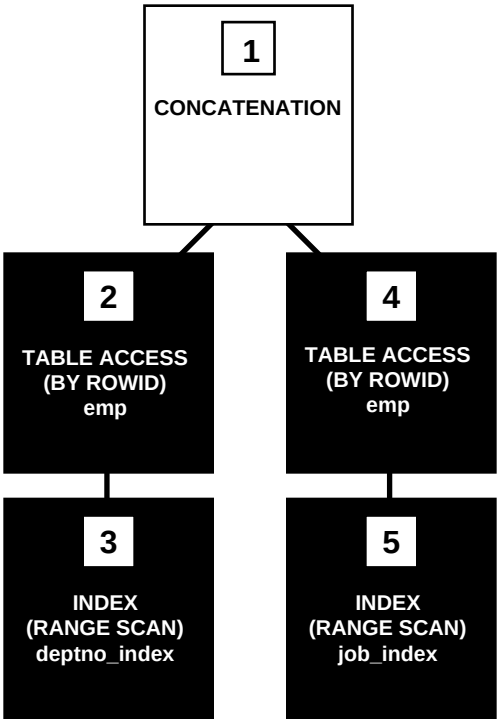
If there are indexes on both the `job` and `deptno` columns, then the optimizer might transform this query into the following equivalent query:

```
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.

The execution plan for the transformed statement might look like the illustration in Figure 2–1.

Figure 2–1 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.

For 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 previous query would result in the following compound query:

```
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.

How the CBO Unnests Subqueries

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 "Understanding Joins" on page 1-42.

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

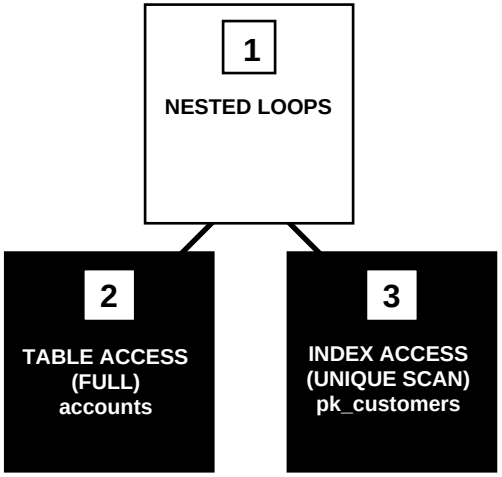
```
SELECT *
  FROM accounts
  WHERE custno IN
    (SELECT custno FROM customers);
```

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 2–2.

Figure 2–2 Execution Plan for a Nested Loops Join



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

> **See Also:** "Understanding Joins" on page 1-42 for information on nested loops joins

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.

How the CBO Merges Views

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 Nonmergeable Views" on page 2-34).

For 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 Nonmergeable 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 nonnull value for a `NULL`.

> **See Also:** "Understanding Joins" on page 1-42

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 "IN Subquery Example" on page 2-35).

Complex merging is not cost-based—it must be enabled with the initialization parameter `OPTIMIZER_FEATURES_ENABLE` or the `MERGE` hint. Without this hint or parameter setting, the optimizer uses another approach (see "How the CBO Pushes Predicates" on page 2-36).

> **See Also:** Chapter 5, "Optimizer Hints" for details about the MERGE and NO_MERGE hints

View with a GROUP BY Clause Example 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.

IN Subquery Example Complex merging can be used for an IN clause with a noncorrelated 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);
```

How the CBO Pushes Predicates

The optimizer can transform a query block that accesses a nonmergeable view by pushing the query block's predicates inside the view's query.

For example:

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.

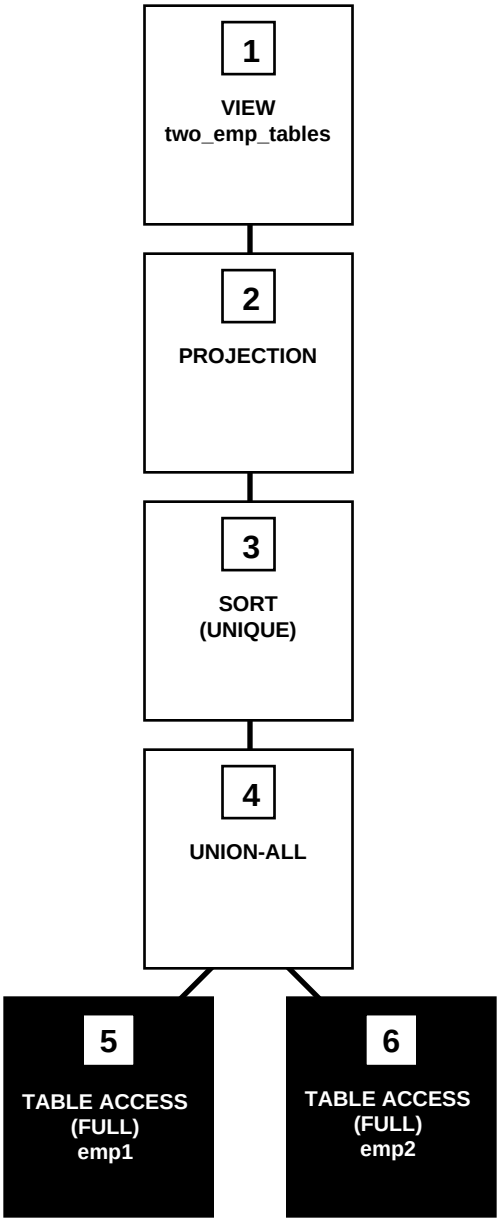
The resulting statement looks like the following:

```
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 2–3 shows the execution plan of the resulting statement.

Figure 2–3 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.

In the following example, 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:

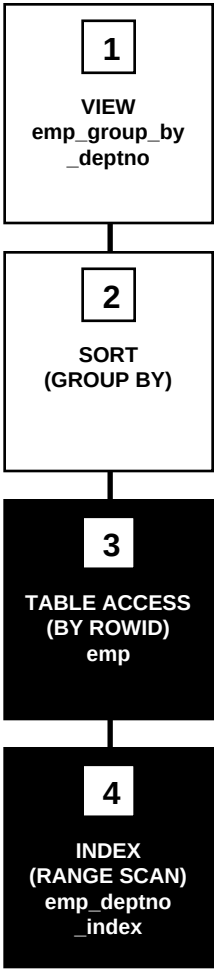
```
SELECT *
  FROM emp_group_by_deptno
  WHERE deptno = 10;
```

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 2–4 shows the execution plan for the resulting statement. The execution plan uses an index on the `deptno` column.

Figure 2–4 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.

How the CBO Applies 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.

For 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:

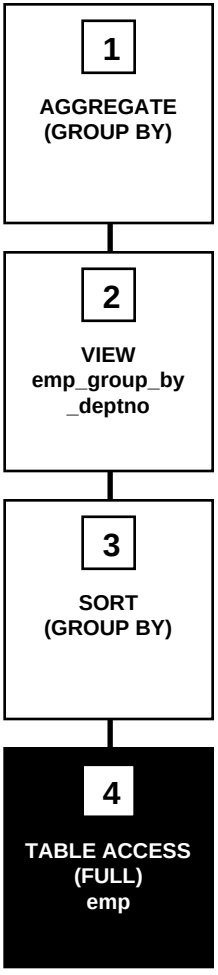
```
SELECT AVG(avg_sal), AVG(min_sal), AVG(max_sal)
    FROM emp_group_by_deptno;
```

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 2–5 shows the execution plan of the resulting statement.

Figure 2–5 Applying Aggregate Functions to a View Defined with GROUP BY Clause



To execute this statement, Oracle performs the following 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.

How the CBO Executes 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.

How the CBO Accesses 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.

For 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 to the name and location of the department in the dept table:

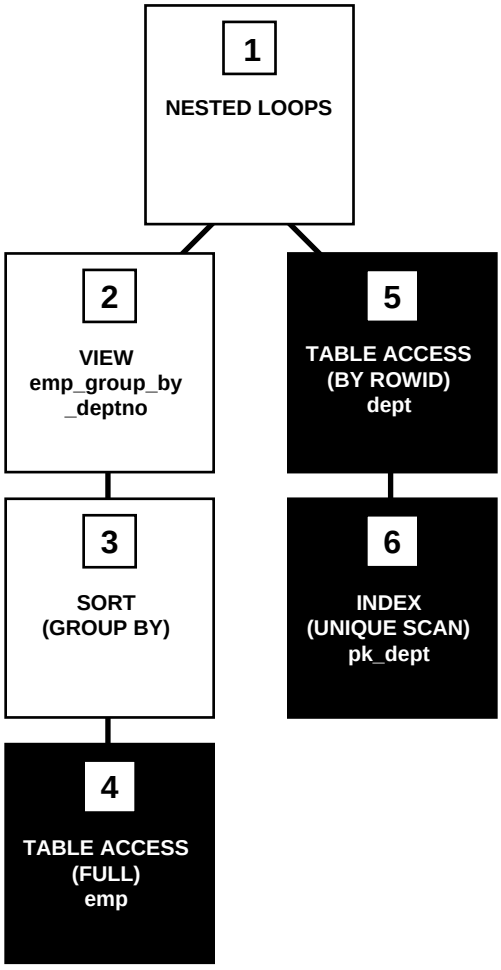
```
SELECT emp_group_by_deptno.deptno, avg_sal, min_sal,
  max_sal, dname, loc
    FROM emp_group_by_deptno, dept
    WHERE emp_group_by_deptno.deptno = dept.deptno;
```

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:** "Understanding Joins" on page 1-42 for more information on how Oracle performs a nested loops join operation

Figure 2–6 shows the execution plan for this statement.

Figure 2–6 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.

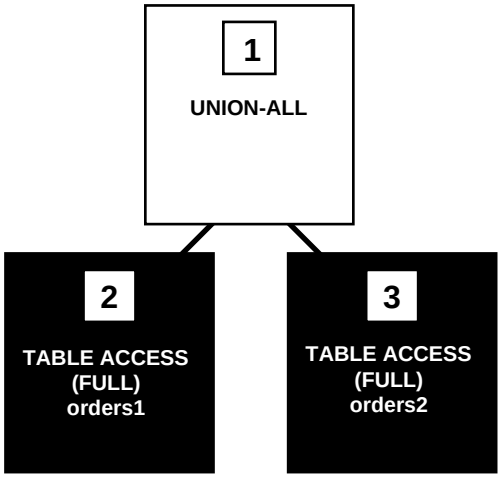
How the CBO Executes Compound Queries

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 2–7 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 2–7 Compound Query with UNION ALL Set Operator



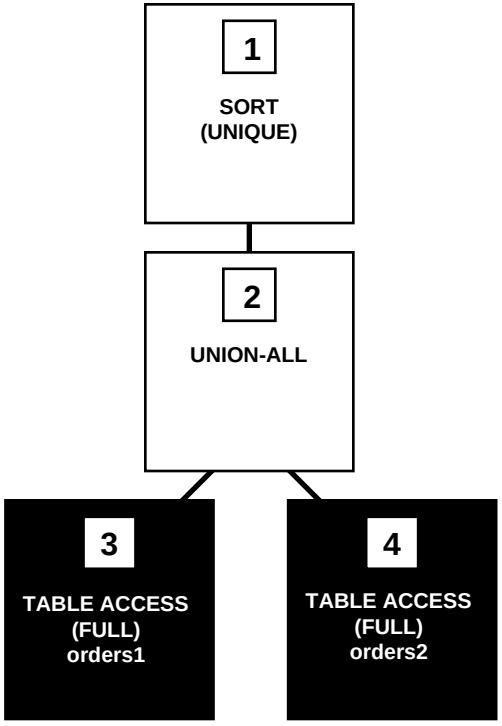
To execute this statement, Oracle performs the following steps:

- 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 2–8 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 2–8 Compound Query with UNION Set Operator

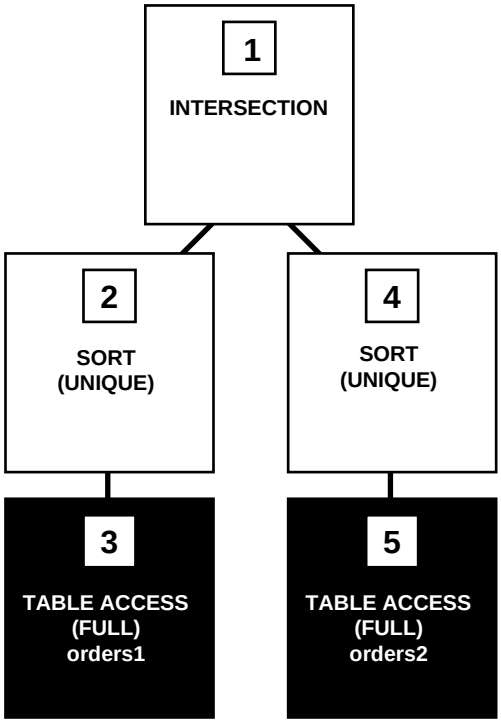


This execution plan is identical to the one for the UNION-ALL operator shown in Figure 2–7 on page 2-47, except that in this case, Oracle uses the SORT operation to eliminate the duplicates returned by the UNION-ALL operation.

Figure 2–9 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 2–9 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.

3

Gathering Optimizer 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

It is possible for the DBA to generate statistics that quantify the data distribution and storage characteristics of tables, columns, indexes, and partitions. The cost-based optimization approach uses these 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.

The statistics are stored in the data dictionary, and they can be exported from one database and imported into another. For example, you might want to transfer your statistics to a test system to simulate your production environment.

Note: The statistics mentioned in this section are CBO statistics, not instance performance statistics visible through V$ views.

You should gather statistics periodically for objects where the statistics become stale over time, due to changing data volumes or changes in column values. 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, collect new statistics on the number of rows. After updating data in a table, it is not necessary to collect new statistics on the number of rows, but you might need new statistics on the average row length.

Statistics should be generated with the DBMS_STATS package.

See Also: *Oracle9i Supplied PL/SQL Packages Reference*

The statistics generated include the following:

- Table statistics
  - Number of rows
  - Number of 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
- System statistics
  - I/O performance and utilization
  - CPU performance and utilization

Generating Statistics

Because the cost-based approach relies on statistics, generate statistics for all tables and clusters and all indexes accessed by your SQL statements before using the cost-based approach. If the size and data distribution of the tables change frequently, then regenerate 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 might 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.

Some statistics are 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.

Oracle Corporation recommends setting the `ESTIMATE_PERCENT` parameter of the `DBMS_STATS` gathering procedures to `DBMS_STATS.AUTO_SAMPLE_SIZE` to maximize performance gains while achieving necessary statistical accuracy. For example, to collect table and column statistics for all tables in `OE` schema with auto-sampling:

```
EXECUTE DBMS_STATS.GATHER_SCHEMA_STATS('OE',DBMS_STATS.AUTO_SAMPLE_SIZE);
```

To estimate statistics, Oracle selects a random sample of data. You can specify the sampling percentage (Oracle Corporation recommends using DBMS_STATS.AUTO_SAMPLE_SIZE) and whether sampling should be based on rows or blocks. When in doubt, use row sampling.

- *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.

Getting Statistics for Partitioned Schema Objects

Partitioned schema objects can contain multiple sets of statistics. They can have statistics that refer to the entire schema object as a whole (global statistics), they can have statistics that refer to an individual partition, and they can have statistics that 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 can seem that generating global statistics from partition-level statistics is straightforward; however, this is true only 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.

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, export, import, and delete statistics. You can also use this package to identify or name statistics gathered.

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. 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 can 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.

See Also:

- *Oracle9i Supplied PL/SQL Packages Reference* for more information about the DBMS_STATS package
- *Oracle9i Data Cartridge Developer's Guide* for more information about user-defined statistics

Gathering Statistics with the DBMS_STATS Package

Table 3–1 lists the procedures in the DBMS_STATS package for gathering statistics:

Table 3–1 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.
GATHER_SYSTEM_STATS	Collects CPU and I/O statistics for the system.

See Also: *Oracle9i Supplied PL/SQL Packages Reference* for syntax and examples of all DBMS_STATS procedures

Gathering System Statistics

System statistics allow the optimizer to consider a system's I/O and CPU performance and utilization. For each plan candidate, the optimizer computes estimates for I/O and CPU costs. It is important to know the system characteristics to pick the most efficient plan with optimal proportion between I/O and CPU cost.

System I/O characteristics depends on many factors and do not stay constant all the time. Using system statistics management routines, database administrators can capture statistics in the interval of time when the system has the most common workload. For example, database applications can process OLTP transactions during the day and run OLAP reports at night. Administrators can gather statistics for both states and activate appropriate OLTP or OLAP statistics when needed. This allows the optimizer to generate relevant costs with respect to available system resource plans.

When Oracle generates system statistics, it analyzes system activity in a specified period of time. Unlike table, index, or column statistics, Oracle does not invalidate already parsed SQL statements when system statistics get updated. All new SQL statements are parsed using new statistics. Oracle Corporation highly recommends that you gather system statistics.

The DBMS_STATS.GATHER_SYSTEM_STATS routine collects system statistics in a user-defined timeframe. You can also set system statistics values explicitly using DBMS_STATS.SET_SYSTEM_STATS. Use DBMS_STATS.GET_SYSTEM_STATS to verify system statistics.

> **Note:** You must have DBA privileges to update dictionary system statistics.

Generating System Statistics Example The following example shows database applications processing OLTP transactions during the day and running reports at night. First, system statistics must be collected.

> **Note:** The values in this example are user-defined. In other words, you must determine an appropriate time interval and name for your environment.

Gather statistics during the day. Gathering ends after 720 minutes and is stored in the `mystats` table:

```
BEGIN
DBMS_STATS.GATHER_SYSTEM_STATS(interval => 720,
                               stattab => 'mystats',
                               statid => 'OLTP');
END;
/
```

Gather statistics during the night. Gathering ends after 720 minutes and is stored in the `mystats` table:

```
BEGIN
DBMS_STATS.GATHER_SYSTEM_STATS(interval => 720,
                               stattab => 'mystats',
                               statid => 'OLAP');
END;
/
```

If appropriate, you can switch between the statistics gathered. It is possible to automate this process by submitting a job to update the dictionary with appropriate statistics.

During the day, the following jobs import the OLTP statistics for the daytime run:

```
VARIABLE jobno number;
BEGIN
   DBMS_JOB.SUBMIT(:jobno,
      'DBMS_STATS.IMPORT_SYSTEM_STATS(''mystats'',''OLTP'');'
      SYSDATE, 'SYSDATE + 1');
   COMMIT;
END;
/
```

During the night, the following jobs import the OLTP statistics for the nighttime run:

```
BEGIN
   DBMS_JOB.SUBMIT(:jobno,
      'DBMS_STATS.IMPORT_SYSTEM_STATS(''mystats'',''OLAP'');'
      SYSDATE + 0.5, 'SYSDATE + 1');
   COMMIT;
END;
/
```

> **See Also:** *Oracle9i Supplied PL/SQL Packages Reference* for detailed information on the procedures in the DBMS_STATS package for implementing system statistics

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.

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 significant changes to the data, then use the DBMS_STATS.GATHER_INDEX_STATS procedure to collect index statistics. The GATHER_INDEX_STATS procedure does not run in parallel.

> **See Also:** *Oracle9i SQL Reference* for more information about the COMPUTE STATISTICS clause

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 (by specifying additional parameters).

> **See Also:** "Preserving Versions of Statistics" on page 3-12

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 in a different statistics table or a statistics table with a different statistics identifier.
2. Use DBMS_STATS.IMPORT_SCHEMA_STATS to restore the old statistics. The application is now ready to run again.

You might 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. Stored outlines are precompiled execution plans that Oracle can use to mimic proven application performance characteristics.

> **See Also:** Chapter 7, "Using Plan Stability"

2. Use DBMS_STATS.IMPORT_SCHEMA_STATS to restore the new statistics. The 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.
GATHER AUTO	Gathers *all* the statistics for the objects of a specific schema (or database with DBMS_STATS.GATHER_DATABASE_STATS()) that are not up-to-date.

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 Before you can utilize automated statistics gathering for a particular table, you must bring either the tables of a specific schema or a complete database into the monitoring mode. Do this with the DBMS_STATS.ALTER_SCHEMA_TABLE_MONITORING or DBMS_STATS.ALTER_DATABASE_TABLE_MONITORING procedures. Or, you can enable the monitoring attribute using the MONITORING keyword. This keyword is part of the CREATE TABLE and ALTER TABLE statement syntax. Monitoring tracks 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. Then, you can enable automated statistics gathering by setting up a recurring job (perhaps by using job queues) that invokes DBMS_STATS.GATHER_TABLE_STATS with the GATHER STALE option at an appropriate interval for your application.

Objects are considered stale when 10% of the total rows have been changed. When you issue GATHER_TABLE_STATS with GATHER STALE, the procedure checks the USER_TAB_MODIFICATIONS view. If a monitored table has been modified more than 10%, then statistics are gathered again. The information about changes of tables, as shown in the USER_TAB_MODIFICATIONS view, can be flushed from the SGA into the data dictionary with the DBMS_STATS.FLUSH_DATABASE_MONITORING_INFO procedure.

Note: There can be a few minutes delay while Oracle propagates information to this view.

To disable monitoring, use the DBMS_STATS.ALTER_SCHEMA_TABLE_MONITORING or DBMS_STATS.ALTER_DATABASE_TABLE_MONITORING procedures, or use the NOMONITORING keyword.

> **See Also:** *Oracle9i SQL Reference* for more information about the CREATE TABLE and ALTER TABLE syntax and the MONITORING and NOMONITORING keywords

Enabling Automated Statistics Gathering The GATHER STALE option only gathers statistics for tables that have stale statistics and for which you have enabled monitoring, as described above.

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 gathering statistics 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 the 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 gather manual 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:** *Oracle9i Supplied PL/SQL Packages Reference* for more information on DBMS_STATS procedures and parameters

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.
- Most importantly, in the future, ANALYZE will not collect statistics needed by the cost-based optimizer.

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.

Note: Currently, the DBMS_STATS package does not call statistics collection methods associated with individual columns. Use the ANALYZE statement.

See Also: *Oracle9i SQL Reference* for detailed information about the ANALYZE statement

Finding Data Distribution

The statistics gathered help you determine how the data is distributed across the tables. The optimizer assumes that the data is uniformly distributed. The actual data distribution in the tables can be easily analyzed by viewing the appropriate dictionary table, as described in DBA_TABLES for tables and DBA_TAB_COL_STATISTICS for column statistics.

Histograms can be used to determine attribute skew.

See Also:

- "Understanding Statistics" on page 3-2
- "Using Histograms" on page 3-20

Missing Statistics

When statistics do not exist, the optimizer uses the following default values. Table 3–2 shows the defaults you can expect when statistics are missing.

Table 3–2 Default Table and Index Values When Statistics are Missing

Statistic	Default Value Used by Optimizer
Tables	
■ Cardinality	100 rows
■ Average row length	20 bytes
■ Number of blocks	100
■ Remote cardinality	2000 rows
■ Remote average row length	100 bytes

Table 3–2 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 * number of blocks)

Using Statistics

This section provides guidelines on how to use and view 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.

Using Statistics Tables

The DBMS_STATS package lets you 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

Use the DBMS_STATS package to view the statistics stored in the data dictionary or in a statistics table. For example:

```
DECLARE
  num_rows NUMBER;
  num_blocks NUMBER;
  avg_row_len NUMBER;

BEGIN
  -- retrieve the values of table statistics on OE.ORDERS
  -- statistics table name: OE.SAVESTATS    statistics ID: TEST1

  DBMS_STATS.GET_TABLE_STATS('OE','ORDERS',null,
    'SAVESTATS','TEST1',
    num_rows,num_blocks,avg_row_len);

  -- print the values

DBMS_OUTPUT.PUT_LINE('num_rows='||num_rows||',num_blocks='||num_blocks||

    ',avg_row_len='||avg_row_len);
END;
```

> **Note:** Statistics held in a statistics table are held in a form that is only understood by using DBMS_STATS package.

Or, query the following data dictionary views for statistics in the data dictionary:

> **Note:** Use the appropriate USER_, ALL_, or DBA_ view. The list below shows the DBA_ views.

- DBA_TABLES
- DBA_TAB_COL_STATISTICS
- DBA_INDEXES
- DBA_CLUSTERS
- DBA_TAB_PARTITIONS
- DBA_TAB_SUBPARTITIONS
- DBA_IND_PARTITIONS
- DBA_IND_SUBPARTITIONS
- DBA_PART_COL_STATISTICS
- DBA_SUBPART_COL_STATISTICS

> **See Also:** *Oracle9i Database Reference* for information on the statistics in these views

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','SO_LAST_ALL');
```

This returns the following typical data:

```
TABLE_NAME                NUM_ROWS   BLOCKS   AVG_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
SO_LAST_ALL
```

> **Note:** Because the table SO_LAST_ALL has no statistics, it returns blanks for all columns.

Verifying Index Statistics

To verify that index statistics are available and assist you in determining which are the best indexes to use in an application, execute the following statement against the dictionary DBA_INDEXES view:

```
SQL> SELECT INDEX_NAME "NAME", NUM_ROWS, DISTINCT_KEYS "DISTINCT",
  1  LEAF_BLOCKS, CLUSTERING_FACTOR "CF", BLEVEL "LEVEL",
  2  AVG_LEAF_BLOCKS_PER_KEY "ALFBPKEY"
  3  FROM DBA_INDEXES
  4  WHERE owner = 'SH'
  5* ORDER BY INDEX_NAME;

NAME                           NUM_ROWS   DISTINCT LEAF_BLOCKS         CF      LEVEL   ALFBPKEY
------------------------------ ---------- ---------- ----------- ---------- ---------- ----------
CUSTOMERS_PK                        50000      50000         454       4405          2          1
PRODUCTS_PK                         10000      10000          90       1552          1          1
PRODUCTS_PROD_CAT_IX                10000          4          99       4422          1         24
PRODUCTS_PROD_SUBCAT_IX             10000         37         170       6148          2          4
SALES_PROD_BIX                       6287        909        1480       6287          1          1
SALES_PROMO_BIX                      4727        459         570       4727          1          1

6 rows selected.
```

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 in the index. This is the number of I/Os needed to find the desired rows of data.
- Clustering factor (CF). This is the collocation amount of the index block relative to data blocks. The higher the CF, the less likely the optimizer is to select this index.
- Average leaf blocks per key (ALFBKEY). Average number of leaf blocks in which each distinct value in the index appears, rounded to the nearest integer. For indexes that enforce UNIQUE and PRIMARY KEY constraints, this value is always one.

Determining if You Have Chosen the Right Index

Use the following notes to assist you in deciding whether you have chosen an appropriate index for a table, data, and query:

DISTINCT Consider index ap_invoices_n3, the number of distinct keys is two. 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_COL_STATISTICS view:

```
SQL> SELECT COLUMN_NAME, NUM_DISTINCT, NUM_NULLS, NUM_BUCKETS, DENSITY
    FROM DBA_TAB_COL_STATISTICS
    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 important for the following conditions:

- 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 previous example.

NUM_DISTINCT Column Statistic

Low The number of distinct values for the columns CONVERTED_FLAG is one. 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 an index unless there is much skew or there are a lot of nulls. 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 is needed on that column in order for the optimizer to recognize the skew and use an index on this column.

High NUM_DISTINCT is more than 1 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 nulls, such as the COST_DISTRIBUTED_FLAG column, and if this column is used as the index, then the resulting data set is large.

High If there are many nulls on a particular column, such as the CONVERTED_FLAG column, and if this column is used as the index, then the resulting data set is small. This means that COST_DISTRIBUTED_FLAG is a more appropriate column to index.

DENSITY Column Statistic

This indicates the density of the values of that column. This is calculated by 1 over `NUM_DISTINCT`.

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

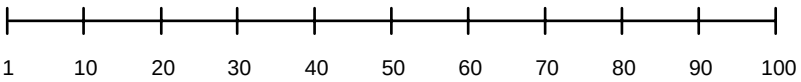
The cost-based optimizer can use data value histograms to get accurate estimates of the distribution of column data. A *histogram* partitions the values in the column into bands, so that all column values in a band fall within the same range. Histograms provide improved selectivity estimates in the presence of data skew, resulting in optimal execution plans with nonuniform data distributions.

One of the fundamental tasks 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. Some 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.

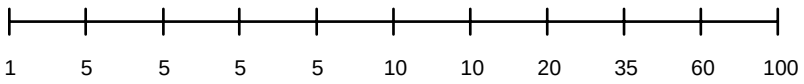
See Also: "Types of Histograms" on page 3-22

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 looks 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. Histogram statistics data is persistent, so the space required to save the data depends on the sample size. In general, create histograms on columns that are used frequently in WHERE clauses of queries and have a 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. You can generate histograms for columns of a table or partition. 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.

Oracle Corporation recommends using the DBMS_STATS package to have the database automatically decide which columns need histograms. This is done by specifying size "auto".

> **See Also:** *Oracle9i Supplied PL/SQL Packages Reference* for more information on the DBMS_STATS package

Choosing the Number of Buckets for a Histogram

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. 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, and the data distribution both affect a histogram's usefulness, you might need to experiment with different numbers of buckets to obtain optimal results.

Types of Histograms

There are two types of histograms:

- Understanding Height-Based Histograms
- Understanding Value-Based Histograms

Understanding 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. Only the last (largest) values in each bucket appear as bucket (end point) values.

Consider that a table's query results in the following four sample values: 4, 18, 30, and 35.

For a height-based histogram, each of these values occupies 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)
```

Understanding Value-Based Histograms

Value-based histograms are created when the number of distinct values is less than or equal to the number of histogram buckets specified. In value-based histograms, all the values in the column have corresponding buckets, and the bucket number reflects the repetition count of each value. These can also be known as frequency histograms.

Consider the same four sample values in the previous example. 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 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 nonrepresentative selectivity value.

Example Using Histograms

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 nonnull 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 previous example, the selectivity, based on the histogram data, is 0.008. This is 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:

> **Note:** Use the appropriate USER_, ALL_, or DBA_ view. The list below shows the DBA_ views.

- DBA_HISTOGRAMS
- DBA_PART_HISTOGRAMS
- DBA_SUBPART_HISTOGRAMS
- DBA_TAB_COL_STATISTICS

Number of Rows View the DBA_HISTOGRAMS dictionary table for the number of buckets (in other words, the number of rows) for each column:

- ENDPOINT_NUMBER
- ENDPOINT_VALUE

> **See Also:** *Oracle9i Database Reference* for column descriptions of data dictionary views, as well as histogram use and restrictions

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"
     ORDER BY ENDPOINT_NUMBER;
```

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 five values are represented in the bucket containing the endpoint 1370. If endpoint numbers are very different, then this implies the use of more buckets, where one row corresponds to one bucket.

4

Understanding Indexes and Clusters

This chapter provides an overview of data access methods that can enhance performance and of situations to avoid.

This chapter contains the following sections:

- Understanding Indexes
- Using Function-based Indexes
- Using Index-Organized Tables
- Using Bitmap Indexes
- Using Bitmap Join Indexes
- Using Domain Indexes
- Using Clusters
- Using Hash Clusters

Understanding Indexes

This section describes the following:

- Tuning the Logical Structure
- Choosing Columns and Expressions to Index
- Choosing Composite Indexes
- Writing Statements that Use Indexes
- Writing Statements that Avoid Using Indexes
- Re-creating Indexes
- Using Nonunique Indexes to Enforce Uniqueness
- Using Enabled Novalidated Constraints

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 write-intensive application. In other words, do not build indexes unless necessary.

To maintain optimal performance, drop indexes that an application is not using. You can find indexes that are not being used by using the `ALTER INDEX MONITORING USAGE` functionality over a period of time that is representative of your workload. This monitoring feature records whether or not an index has been used. If you find that an index has not been used, then drop it. Be careful to select a representative workload to monitor.

See Also: *Oracle9i SQL Reference*

Indexes within an application sometimes have uses that are not immediately apparent from a survey of statement execution plans. An example of this is a foreign key index on a parent table which prevents share locks being taken out on a child table.

If you are deciding whether to 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 can 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 can choose to use that index for other statements in the application as well. For this reason, reexamine the application's performance and rerun the SQL trace facility after you have tuned those statements that you initially identified for tuning.

Note: You can use the Oracle Index Tuning Wizard to detect tables with inefficient indexes. The Oracle Index Tuning wizard is an Oracle Enterprise Manager integrated application available with the Oracle Tuning Pack. Similar functionality is available from the Virtual Index Advisor (a feature of SQL Analyze) and Oracle Expert. For more information, see the *Database Tuning with the Oracle Tuning Pack* manual.

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 used frequently in `WHERE` clauses.
- Consider indexing keys that are used frequently to join tables in SQL statements. For more information on optimizing joins, see the section "Using Hash Clusters" on page 4-20.
- 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.

Indexing low selectivity columns can be helpful if the data distribution is skewed so that one or two values occur much less often than the others.

- 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 where the index is modified frequently.
- Do not index columns that are modified frequently. `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.
- 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 (except with function-based indexes).
- 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 `UPDATE`s and `DELETE`s 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 `INSERT`s, `UPDATE`s, and `DELETE`s and the use of the space required to store the index. You might want to experiment by comparing the processing times of the SQL statements with and without indexes. You can measure processing time with the SQL trace facility.

> **See Also:** *Oracle9i Application Developer's Guide - Fundamentals* for more information on the effects of foreign keys on locking

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 higher 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.

> **Note:** This is no longer the case with index skip scans. See "Index Skip Scans" on page 1-35.

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 used together frequently 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 allow the CBO the option of using an index access path, ensure that the statement contains a construct that makes such an access path available.

Writing Statements that Avoid Using Indexes

In some cases, you might want to prevent a SQL statement from using an access path that uses an existing index. You might 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.

> **See Also:** Chapter 5, "Optimizer Hints" for more information on the `NO_INDEX`, `FULL`, `INDEX`, `INDEX_COMBINE`, and `AND_EQUAL` hints

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 might be better to use sequential index lookup than parallel table scan.

Re-creating Indexes

You might 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 might use the existing index instead of the base table to improve the performance of the index build.

> **Note:** Remember to include the CREATE STATISTICS statement on the CREATE or REBUILD to avoid calling DBMS_STATS after the index creation or rebuild. You can use the Oracle Enterprise Manager Reorg Wizard to identify indexes that require rebuilding. The Reorg Wizard can also be used to rebuild the indexes.

However, there are cases where it can 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 can 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.

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).

Usually, ALTER INDEX ... REBUILD is 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:** *Oracle9i SQL Reference* for more information about the CREATE INDEX and ALTER INDEX statements, as well as restrictions on rebuilding indexes

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 reuse. You can also rebuild the index online.

> **See Also:** *Oracle9i SQL Reference* and *Oracle9i Database Administrator's Guide* for more information about the syntax for this statement

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 for new data. 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 can be unnamed and should be created enabled and validated. All other constraints (CHECK,

UNIQUE, PRIMARY KEY, and FOREIGN KEY) should be named and "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:** *Oracle9i Database Concepts* for a complete discussion of integrity constraints

Using Function-based Indexes

A function-based index includes columns that are either transformed by a function (for example, the UPPER function), or columns that are included in an expression (for example, col1 + col2).

Defining a function-based index on the transformed column or expression allows that data to be returned using the index when that function or expression is used in

a WHERE clause or an ORDER BY clause. Therefore, a function-based index can be beneficial when frequently-executed SQL statements include transformed columns (or columns in expressions) in a WHERE (or ORDER BY) clause.

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';
```

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.

If the QUERY_REWRITE_INTEGRITY parameter is set to ENFORCED (the default), then Oracle uses function-based indexes to derive values of SQL expressions only. This also includes SQL functions. If QUERY_REWRITE_INTEGRITY is set to any value other than ENFORCED, then Oracle uses the function-based index, even if it is based on a user-defined (rather than SQL) function.

Function-based indexes are an efficient mechanism for evaluating statements that contain functions in WHERE clauses. You can create a function-based index to store 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;
```

You can also use function-based indexes for linguistic 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:

- *Oracle9i Application Developer's Guide - Fundamentals* for detailed information on using function-based indexes
- *Oracle9i SQL Reference* for more information on the `CREATE INDEX` statement

Using Index-Organized Tables

An index-organized table differs from an ordinary table in that the data for the table is held in its associated index. Changes to the table data, such as adding new rows, updating rows, or deleting rows, result only in updating the index. Because data rows are stored in the index, index-organized tables provide faster key-based access to table data for queries that involve exact match or range search, or both.

See Also: *Oracle9i Database Concepts*

Using Bitmap Indexes

This section describes:

- When to Use Bitmap Indexes
- Using Bitmap Indexes with Good Performance
- Initialization Parameters for Bitmap Indexing
- Using Bitmap Access Plans on Regular B-tree Indexes
- Bitmap Index Restrictions

See Also: *Oracle9i Database Concepts* and *Oracle9i Data Warehousing Guide* for more information on bitmap indexing

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 for Bitmap Indexes
- Comparing B-Tree Indexes to Bitmap Indexes
- Maintenance Considerations for Bitmap Indexes

Performance Considerations for Bitmap Indexes

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:** *Oracle9i Database Concepts* for more information on optimizing anti-joins and semi-joins

Comparing B-Tree Indexes to Bitmap Indexes

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 are commonly 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, keep B-tree indexes on high-cardinality columns while creating bitmap indexes on low-cardinality columns.

Maintenance Considerations for Bitmap Indexes

Bitmap indexes benefit data warehousing applications, but they are not appropriate for OLTP applications with a heavy load of concurrent `INSERT`s, `UPDATE`s, and `DELETE`s. In a data warehousing environment, data is maintained usually 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 previously are regular sorts and use the regular sort area, determined by `SORT_AREA_SIZE`. The `BITMAP_MERGE_AREA_SIZE` and `CREATE_BITMAP_AREA_SIZE` initialization parameters described in "Initialization Parameters for Bitmap Indexing" on page 4-16 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 can be smaller). The locking granularity is one such bitmap segment. This can 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 can 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.

Using Bitmap Indexes with Good Performance

Using Hints with Bitmap Indexes

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 might 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 nonhinted 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 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:** Chapter 5, "Optimizer Hints" for more information on the INDEX_COMBINE hint

Performance Tips for Bitmap Indexes

To get optimal performance and disk space usage with bitmap indexes, consider the following tips:

- 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:** Chapter 9, "Using EXPLAIN PLAN" for more information about bitmap EXPLAIN PLAN output

Mapping Bitmaps to Rowids Efficiently

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:**
>
> - "Using Bitmap Indexes" on page 4-11 for more information
> - *Oracle9i SQL Reference* for syntax on MINIMIZE and NOMINIMIZE

Using Bitmap Indexes on Index-Organized Tables

The rowids used in bitmap indexes on index-organized tables are in a mapping table, not the base table. The mapping table maintains a mapping of logical rowids (needed to access the index-organized table) to physical rowids (needed by the

bitmap index code). There is one mapping table per index-organized table, and it is used by all the bitmap indexes created on the index-organized table.

Note: Moving rows in an index-organized table does not make the bitmap indexes built on that index-organized table unusable.

See Also: *Oracle9i Database Concepts* for information on bitmap indexes and index-organized tables

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 looks like the following:

```
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) is left out because it is not needed.

Initialization Parameters for Bitmap Indexing

The following initialization parameters affect various aspects of using bitmap indexes:

CREATE_BITMAP_AREA_SIZE: memory allocated for bitmap creation

BITMAP_MERGE_AREA_SIZE: memory used to merge bitmaps from an index range scan

SORT_AREA_SIZE: memory used when inserting or updating bitmap indexes

> **See Also:** *Oracle9i Database Reference* for more information on these parameters

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 can 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.

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 Bitmap Join Indexes

In addition to a bitmap index on a single table, you can create a bitmap join index, which is a bitmap index for the join of two or more tables. A bitmap join index is a space efficient way of reducing the volume of data that must be joined by performing restrictions in advance. For each value in a column of a table, a bitmap join index stores the rowids of corresponding rows in another table. In a data warehousing environment, the join condition is an equi-inner join between the primary key column or columns of the dimension tables and the foreign key column or columns in the fact table.

Bitmap join indexes are much more efficient in storage than materialized join views, an alternative for materializing joins in advance. This is because the materialized join views do not compress the rowids of the fact tables.

> **See Also:** *Oracle9i Data Warehousing Guide* for examples and restrictions of bitmap join indexes

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: *Oracle Spatial User's Guide and Reference* for information about the `SpatialIndextype`

Using Clusters

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

See Also: *Oracle9i Database Concepts* for more information on clusters

Follow these guidelines when deciding whether to cluster tables:

- Cluster tables that are accessed frequently by the application in join statements.
- Do not cluster tables if the 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 might need to migrate the modified row to another block to maintain the cluster.

- Do not cluster tables if the 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. An alternative is to use an index organized 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 the application. For example, you might decide that the performance gain for join statements outweighs the performance loss for statements that modify cluster key values. You might want to experiment and compare processing times with the tables both clustered and stored separately. To create a cluster, use the CREATE CLUSTER statement.

> **See Also:** *Oracle9i Application Developer's Guide - Fundamentals* for more information on creating clusters

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 the application. You might 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:

Follow these guidelines for choosing when to use hash clusters:

- Use hash clusters to store tables accessed frequently 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 store a table in a hash cluster if the 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 might 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 the 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 might need 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 joined frequently with other tables, provided that hashing is appropriate for the table based on the previous points in this list.

5

Optimizer Hints

This chapter explains how to use hints to force various approaches.

This chapter contains the following sections:

- Understanding Optimizer Hints
- Using Optimizer Hints

Understanding Optimizer Hints

As an application designer, you might know information about your data that the optimizer does not know. For example, you might know that a certain index is more selective for certain queries. Based on this information, you might 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 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 path
- 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:** Chapter 3, "Gathering Optimizer Statistics" for more information on default values

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:** *Oracle9i SQL Reference* for more information on comments

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 always follows the INSERT keyword, and the PARALLEL hint can follow the INSERT keyword.

The syntax below shows hints contained in both styles of comments that Oracle supports within a statement block.

```
{DELETE|INSERT|SELECT|UPDATE} /*+ hint [text] [hint[text]]... */
```

or

```
{DELETE|INSERT|SELECT|UPDATE} --+ hint [text] [hint[text]]...
```

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 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. 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:** "Using Optimizer Hints" on page 5-6 for the syntax of each hint

Specifying a Full Set of Hints

When using hints, in some cases, you might need to specify a full set of hints in order 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 might not necessarily use that hint, because the optimizer might 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 a 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 might not be honored from a query against the view.

Local vs. Global Hints

Table hints (in other words, hints that specify a table) generally refer to tables in the DELETE, SELECT, or UPDATE statement in which the hint occurs, not to tables inside any views referenced by the statement. When you want to specify hints for tables that appear inside views, Oracle recommends using global hints instead of embedding the hint in the view. 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:** "Global Hints" on page 5-39 for information on how to create global hints

Note: The SQL Analyze tool (available with the Oracle Tuning Pack), provides a graphical user interface for working with optimizer hints. The Hint Wizard (a feature of SQL Analyze) helps you easily add or modify hints in SQL statements. For more information on Oracle SQL Analyze, see the *Database Tuning with the Oracle Tuning Pack* manual.

Using Optimizer 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(n)
- 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).

all_rows_hint::=

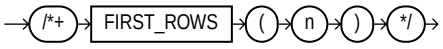
For example, the optimizer uses the cost-based approach to optimize this statement for best throughput:

```
SELECT /*+ ALL_ROWS */ empno, ename, sal, job
FROM emp
WHERE empno = 7566;
```

FIRST_ROWS(*n*)

The hints FIRST_ROWS(*n*) (where *n* is any positive integer) or FIRST_ROWS instruct Oracle to optimize an individual SQL statement for fast response. FIRST_ROWS(n) affords greater precision, because it instructs Oracle to choose the plan that returns the first *n* rows most efficiently. The FIRST_ROWS hint, which optimizes for the best plan to return the first single row, is retained for backward compatibility and plan stability.

first_rows_hint::=



For example, the optimizer uses the cost-based approach to optimize this statement for best response time:

```
SELECT /*+ FIRST_ROWS(10) */ empno, ename, sal, job
FROM emp
WHERE deptno = 20;
```

In this example each department contains many employees. The user wants the first 10 employees of department #20 to be displayed as quickly as possible.

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.

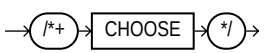
These estimates might not be as accurate as those gathered by the DBMS_STATS package. Therefore, use the DBMS_STATS package to gather statistics. If you specify hints for access paths or join operations along with either the ALL_ROWS or FIRST_ROWS hint, then the optimizer gives precedence to the access paths and join operations specified by the hints.

> **See Also:** "How the CBO Optimizes SQL Statements for Fast Response" on page 1-13 for an explanation of the difference between FIRST_ROWS(*n*) and FIRST_ROWS.

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.

choose_hint::=



For example:

```
SELECT /*+ CHOOSE */ empno, ename, sal, job
FROM emp
WHERE empno = 7566;
```

RULE

rule_hint::=

For example:

```
SELECT --+ RULE
empno, ename, sal, job
FROM emp
WHERE empno = 7566;
```

Hints for Access Paths

Each hint described in this section suggests an access path for a table.

- FULL
- ROWID
- CLUSTER
- HASH
- INDEX
- INDEX_ASC
- INDEX_COMBINE
- INDEX_JOIN
- INDEX_DESC
- INDEX_FFS
- NO_INDEX
- AND_EQUAL

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.

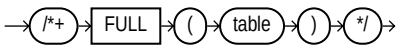
You must specify the table to be accessed exactly as it appears in the statement. If the statement uses an alias for the table, then use the alias rather than the table name in the hint. The table name within the hint should not include the schema name if the schema name is present in the statement.

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 *Oracle9i Database Concepts* and *Oracle9i Database Reference.*

FULL

The FULL hint explicitly chooses a full table scan for the specified table.

full_hint::=



where table specifies the name or alias of the table on which the full table scan is to be performed. If the statement does not use aliases, then the table name is the default alias.

For example:

```
SELECT /*+ FULL(A) don't use the index on accno */ accno, bal
FROM accounts a
WHERE accno = 7086854;
```

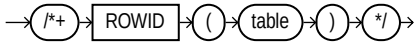
Oracle performs a full table scan on the accounts table to execute this statement, even if there is an index on the accno column that is made available by the condition in the WHERE clause.

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.

rowid_hint::=



where table specifies the name or alias of the table on which the table access by rowid is to be performed.

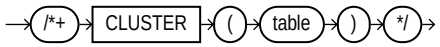
For example:

```
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.

cluster_hint::=



where table specifies the name or alias of the table to be accessed by a cluster scan.

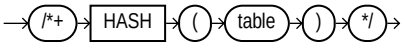
For example:

```
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.

hash_hint::=

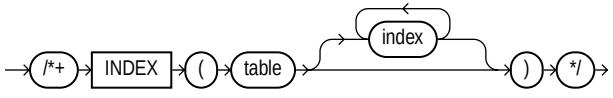


where `table` specifies the name or alias of the table to be accessed by a hash scan.

INDEX

The `INDEX` hint explicitly chooses an index scan for the specified table. You can use the `INDEX` hint for domain, B-tree, bitmap, and bitmap join indexes. However, Oracle recommends using `INDEX_COMBINE` rather than `INDEX` for bitmap indexes, because it is a more versatile hint.

index_hint::=



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 can 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 can 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 can 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 the 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:

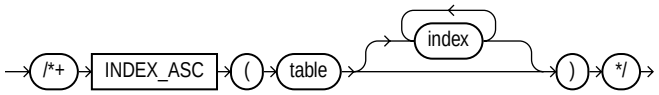
```
SELECT /*+ INDEX(patients sex_index) use sex_index because there are few
    male patients  */ name, height, weight
FROM patients
WHERE sex = 'm';
```

The INDEX hint applies to IN-list predicates; it forces the optimizer to use the hinted index, if possible, for an IN-list predicate. Multicolumn 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.

index_asc_hint::=



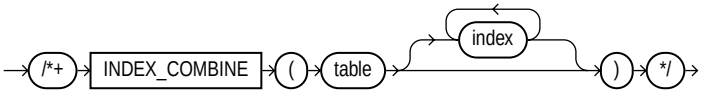
Each parameter serves the same purpose as in the INDEX hint.

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 might 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.

index_combine_hint::=



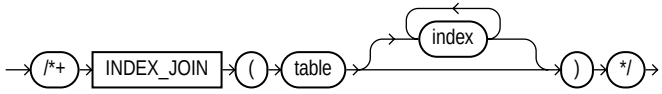
For example:

```
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.

index_join_hint::=



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.

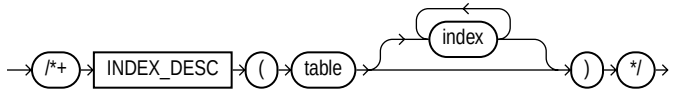
For example:

```
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. In a partitioned index, the results are in descending order within each partition.

index_desc_hint::=

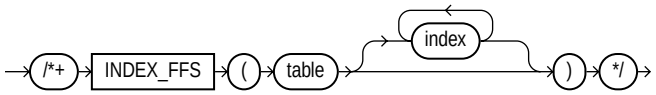


Each parameter serves the same purpose as in the INDEX hint.

INDEX_FFS

The INDEX_FFS hint causes a fast full index scan to be performed rather than a full table scan.

index_ffs_hint::=



For example:

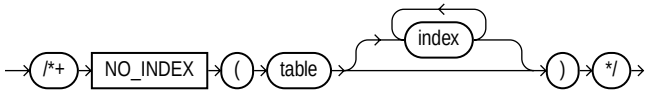
```
SELECT /*+INDEX_FFS(emp emp_empno)*/ empno
FROM emp
WHERE empno > 200;
```

See Also: "Full Scans" on page 1-36

NO_INDEX

The NO_INDEX hint explicitly disallows a set of indexes for the specified table.

no_index_hint::=



- 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.

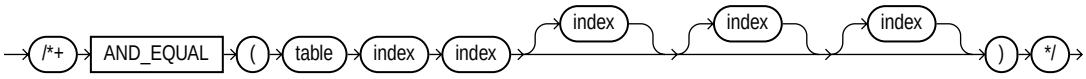
For example:

```
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.

and_equal_hint::=



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.

Hints for Query Transformations

Each hint described in this section suggests a SQL query transformation.

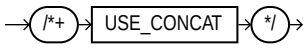
- USE_CONCAT
- NO_EXPAND
- REWRITE
- NOREWRITE
- MERGE
- NO_MERGE
- STAR_TRANSFORMATION
- FACT
- NO_FACT

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. Generally, 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.

use_concat_hint::=



For 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 that the cost is lower than not using it.

no_expand_hint::=

For example:

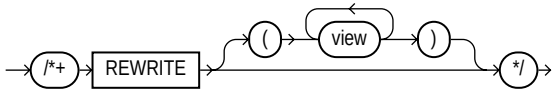
```
SELECT /*+NO_EXPAND*/ *
FROM emp
WHERE empno = 50 OR empno = 100;
```

REWRITE

The REWRITE hint forces the cost-based optimizer to rewrite a query in terms of materialized views, when possible, without cost consideration. 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.

rewrite_hint::=

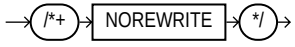


> **See Also:** *Oracle9i Database Concepts* and *Oracle9i Application Developer's Guide - Fundamentals* for more information on materialized views

NOREWRITE

The NOREWRITE hint disables query rewrite for the query block, overriding the setting of the parameter QUERY_REWRITE_ENABLED. Use the NOREWRITE hint on any query block of a request.

norewrite_hint::=



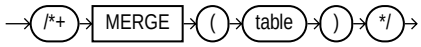
MERGE

The MERGE hint lets you merge a view on a per-query basis.

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_hint::=



For example:

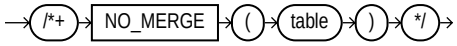
```
SELECT /*+MERGE(v)*/ e1.ename, e1.sal, v.avg_sal
FROM emp e1,
  (SELECT deptno, avg(sal) avg_sal
   FROM emp e2
   GROUP BY deptno) v
WHERE e1.deptno = v.deptno AND e1.sal > v.avg_sal;
```

Note: This example requires complex view merging to be enabled.

NO_MERGE

The NO_MERGE hint causes Oracle not to merge mergeable views.

no_merge_hint::=



This hint lets the user have more influence over the way in which the view is accessed.

For 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 `dallasdept` 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.

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.

star_transformation_hint::=

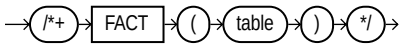
See Also:

- *Oracle9i Database Concepts* for a full discussion of star transformation.
- *Oracle9i Database Reference* for more information on the STAR_TRANSFORMATION_ENABLED initialization parameter.

FACT

The FACT hint is used in the context of the star transformation to indicate to the transformation that the hinted table should be considered as a fact table.

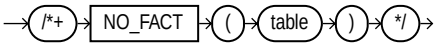
fact_hint::=



NO_FACT

The NO_FACT hint is used in the context of the star transformation to indicate to the transformation that the hinted table should not be considered as a fact table.

no_fact_hint::=



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.

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 might 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 lets you choose an inner and outer table better than the optimizer could.

ordered_hint::=

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;
```

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.

star_hint::=

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:** *Oracle9i Database 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, MERGE_AJ, and NL_AJ
- HASH_SJ, MERGE_SJ, and NL_SJ

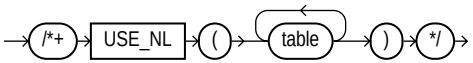
You must specify a table to be joined exactly as it appears in the statement. If the statement uses an alias for the table, then you must use the alias rather than the table name in the hint. The table name within the hint should not include the schema name if the schema name is present in the statement.

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.

use_nl_hint::=



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 might 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:

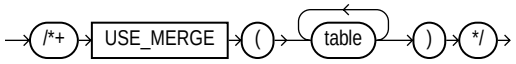
```
SELECT /*+ ORDERED USE_NL(customers) to get first row faster */
accounts.balance, customers.last_name, customers.first_name
FROM accounts, customers
WHERE accounts.custno = customers.custno;
```

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.

use_merge_hint::=



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.

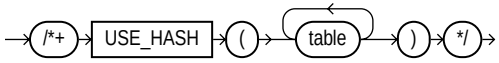
For example:

```
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.

use_hash_hint::=



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.

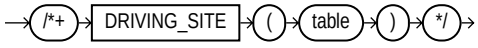
For example:

```
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.

driving_site_hint::=



where `table` is the name or alias for the table at which site the execution should take place.

For 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.

LEADING

The `LEADING` hint causes Oracle to use the specified table 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.

leading_hint::=

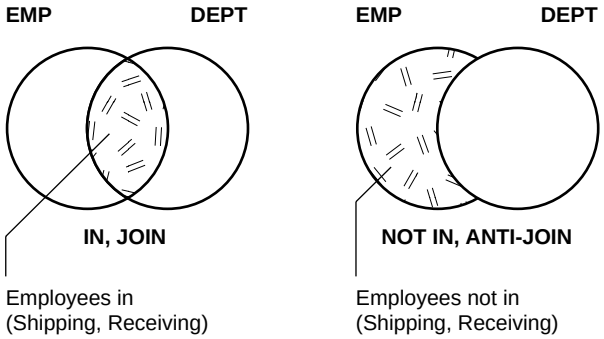
where *table* is the name or alias of a table to be used as the first table in the join order.

HASH_AJ, MERGE_AJ, and NL_AJ

For a specific query, place the `MERGE_AJ`, `HASH_AJ`, or `NL_AJ` hint into the `NOT IN` subquery. `MERGE_AJ` uses a sort-merge anti-join, `HASH_AJ` uses a hash anti-join, and `NL_AJ` uses a nested loop anti-join.

As illustrated in Figure 5–1, 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 5–1 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.

HASH_SJ, MERGE_SJ, and NL_SJ

For a specific query, place the HASH_SJ, MERGE_SJ, or NL_SJ hint into the EXISTS subquery. HASH_SJ uses a hash semi-join, MERGE_SJ uses a sort merge semi-join, and NL_SJ uses a nested loop 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.

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
- PARALLEL_INDEX
- NOPARALLEL_INDEX

> **See Also:** *Oracle9i Data Warehousing Guide* for more information on parallel execution

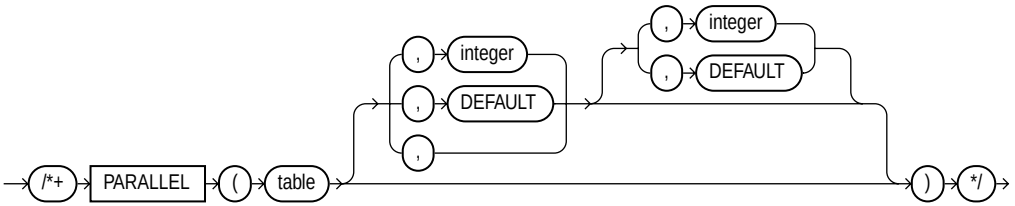
PARALLEL

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.

parallel_hint::=



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, and the second value specifies how the table is to be split among the Oracle Real Application Cluster instances. 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;
```

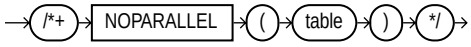
In the next example, the PARALLEL hint overrides the degree of parallelism specified in the emp table definition and tells the optimizer to use the default degree of parallelism determined by the initialization parameters. This hint also specifies that the table should be split among all of the available 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

The NOPARALLEL hint overrides a PARALLEL specification in the table clause. In general, hints take precedence over table clauses.

noparallel_hint::=



The following example illustrates the NOPARALLEL hint:

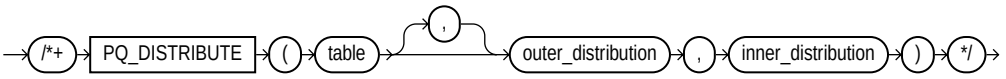
```
SELECT /*+ NOPARALLEL(scott_emp) */ ename
FROM scott.emp scott_emp;
```

PQ_DISTRIBUTE

The PQ_DISTRIBUTE hint improves 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.

pq_distribute_hint::=



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.

> **See Also:** *Oracle9i Database Concepts* for more information on how Oracle parallelizes join operations

There are six combinations for table distribution. Only a subset of distribution method combinations for the joined tables is valid, as explained in Table 5–1.

Table 5–1 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 general rule 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 general rule is: *Use the None/Broadcast hint if the size of the inner table * number of query servers < size of the outer table.*

Table 5–1 Distribution Hint Combinations

Distribution	Interpretation
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.

For example: 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;
```

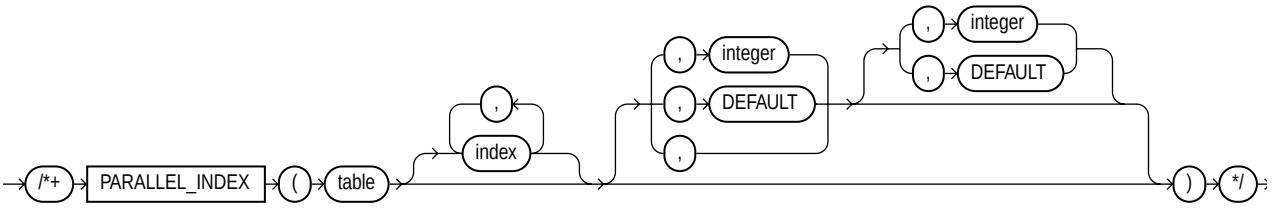
To broadcast the outer table r, the query is:

```
SELECT <column list> /*+ORDERED PQ_DISTRIBUTE(s BROADCAST, NONE) USE_HASH (s) */
FROM r,s
WHERE r.c=s.c;
```

PARALLEL_INDEX

The PARALLEL_INDEX hint specifies the desired number of concurrent servers that can be used to parallelize index range scans for partitioned indexes.

parallel_index_hint::=



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 Oracle Real Application Cluster instances. 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.

For example:

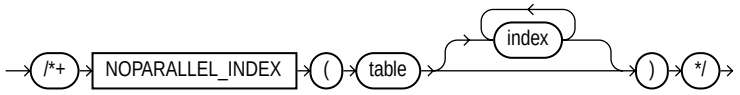
```
SELECT /*+ PARALLEL_INDEX(table1, index1, 3, 2) +/
```

In this example, there are three parallel execution processes to be used on each of two instances.

NOPARALLEL_INDEX

The `NOPARALLEL_INDEX` hint overrides a `PARALLEL` attribute setting on an index to avoid a parallel index scan operation.

noparallel_index_hint::=



Additional Hints

Several additional hints are included in this section:

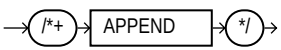
- APPEND
- NOAPPEND
- CACHE
- NOCACHE
- UNNEST
- NO_UNNEST
- PUSH_PRED
- NO_PUSH_PRED
- PUSH_SUBQ
- ORDERED_PREDICATES
- CURSOR_SHARING_EXACT

APPEND

The APPEND hint lets you enable direct-path INSERT if your database is running in serial mode. (Your database is in serial mode if you are not using Enterprise Edition. Conventional INSERT is the default in serial mode, and direct-path INSERT is the default in parallel mode).

In direct-path INSERT, data is appended to the end of the table, rather than using existing space currently allocated to the table. In addition, direct-path INSERT bypasses the buffer cache and ignores integrity constraints. As a result, direct-path INSERT can be considerably faster than conventional INSERT.

append_hint::=



NOAPPEND

The NOAPPEND hint enables conventional INSERT by disabling parallel mode for the duration of the INSERT statement. (Conventional INSERT is the default in serial mode, and direct-path INSERT is the default in parallel mode).

noappend_hint::=

CACHE

The CACHE hint specifies that the blocks retrieved for the 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.

cache_hint::=

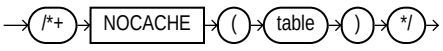
In the following example, the CACHE hint overrides the table's default caching specification:

```
SELECT /*+ FULL (scott_emp) CACHE(scott_emp) */ ename
FROM scott.emp scott_emp;
```

NOCACHE

The NOCACHE hint specifies that the blocks retrieved for the 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.

nocache_hint::=



For 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.

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.

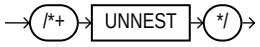
UNNEST_SUBQUERY first verifies if the statement is valid. If the statement is not valid, then subquery unnesting cannot proceed. The statement must then must pass a heuristic test.

If the UNNEST_SUBQUERY parameter is set to true, then the UNNEST hint checks the subquery block for validity only. If it is valid, then subquery unnesting is enabled without Oracle checking the heuristics.

See Also:

- *Oracle9i SQL Reference* for more information on unnesting nested subqueries and the conditions that make a subquery block valid
- Chapter 6, "Optimizing SQL Statements" for more information on the UNNEST_SUBQUERY parameter and managing views

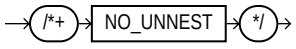
unnest_hint::=



NO_UNNEST

If you enabled subquery unnesting with the UNNEST_SUBQUERY parameter, then the NO_UNNEST hint turns it off for specific subquery blocks.

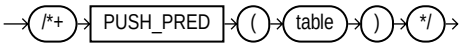
no_unnest_hint::=



PUSH_PRED

The PUSH_PRED hint forces pushing of a join predicate into the view.

push_pred_hint::=



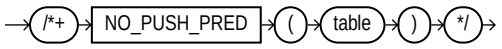
For example:

```
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

The NO_PUSH_PRED hint prevents pushing of a join predicate into the view.

no_push_pred_hint::=

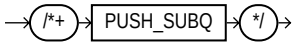


PUSH_SUBQ

The PUSH_SUBQ hint causes non-merged subqueries to be evaluated at the earliest possible place in the execution plan. Generally, 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.

This hint has no effect if the subquery is applied to a remote table or one that is joined using a merge join.

push_subq_hint::=



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.

---

ordered_predicates_hint::=

See Also: *Oracle9i Database Concepts*

CURSOR_SHARING_EXACT

Oracle can replace literals in SQL statements with bind variables if it is safe to do so. This is controlled with the CURSOR_SHARING startup parameter. The CURSOR_SHARING_EXACT hint causes this behavior to be switched off. In other words,

Oracle executes the SQL statement without any attempt to replace literals by bind variables.

cursor_sharing_exact_hint::=

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.

If 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 Path and Join Hints on Views Access path and join hints on referenced views are ignored unless the view contains a single table (or references an Additional Hints view with a single table). For such single-table views, an access path hint or a join hint on the view applies to the table inside the view.

Access Path and Join Hints Inside Views Access path 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 path 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 path 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 applied recursively 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 nonmergeable views, optimization approach and goal hints inside the view are ignored: the top-level query decides the optimization mode.

Because nonmergeable views are optimized separately from the top-level query, access path and join hints inside the view are preserved. For the same reason, access path 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 nonmergeable view is similar to a table.

Global Hints

Table hints (in other words, 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 referenced by the statement. When you want to specify hints for tables that appear inside views, use global hints instead of embedding the hint in the view.

You can transform any table hint in this chapter into a global hint by using an extended syntax for the table name, described as follows.

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.

6

Optimizing SQL Statements

This chapter describes how to identify high-resource SQL statements, explains what should be collected, and provides tuning suggestions.

This chapter contains the following sections:

- Goals for Tuning
- Identifying and Gathering Data on Resource-Intensive SQL
- Overview of SQL Statement Tuning

Goals for Tuning

The objective of tuning is to reduce the response time for end users of the system. This can be accomplished in several ways:

- Reduce the Workload
- Balance the Workload

> **Note:** SQL Analyze can be used for identifying resource intensive SQL statements, generating explain plans, and evaluating SQL performance. For more information on Oracle SQL Analyze, see the *Database Tuning with the Oracle Tuning Pack* manual.

Reduce the Workload

This is what commonly constitutes SQL tuning: finding more efficient ways to process the same workload. It is possible to change the execution plan of the statement without altering the functionality to reduce the resource consumption.

Two examples of how resource usage can be reduced are:

1. If a commonly executed query needs to access a small percentage of data in the table, then it can be executed more efficiently by using an index. By creating such an index, you reduce the amount of resources used.
2. If a user is looking at the first twenty rows of the 10,000 rows returned in a specific sort order, and if the query (and sort order) can be satisfied by an index, then the user does not need to access and sort the 10,000 rows to see the first 20 rows.

Balance the Workload

Systems often tend to have peak usage in the daytime when real users are connected to the system, and low usage in the nighttime. If noncritical reports and batch jobs can be scheduled to run in the nighttime and their concurrency during day time reduced, then it frees up resources for the more critical programs in the day.

Parallelize the Workload

Queries that access large amounts of data (typical data warehouse queries) often can be parallelized. This is extremely useful for reducing the response time in low

concurrency data warehouse. However, for OLTP environments, which tend to be high concurrency, this can adversely impact other users by increasing the overall resource usage of the program.

Identifying and Gathering Data on Resource-Intensive SQL

This section describes the steps involved in identifying and gathering data on poorly-performing SQL statements.

Identifying Resource-Intensive SQL

The first step in identifying resource-intensive SQL is to categorize the problem you are attempting to fix: is the problem specific to a single program (or small number of programs), or is the problem generic over the application?

Tuning a Specific Program

If you are tuning a specific program (GUI or 3GL), then identifying the SQL to examine is a simple matter of looking at the SQL executed within the program.

If it is not possible to identify the SQL (for example, the SQL is generated dynamically), then use SQL_TRACE to generate a trace file that contains the SQL executed, then use TKPROF to generate an output file.

The SQL statements in the TKPROF output file can be ordered by various parameters, such as the execution elapsed time (exeela), which usually assists in the identification by placing the SQL poorly responding SQL at the top of the file. This makes the job of identifying the poorly performing SQL easier if there are many SQL statements in the file.

See Also: Chapter 10, "Using SQL Trace and TKPROF"

Tuning an Application / Reducing Load

If your whole application is performing suboptimally, or if you are attempting to reduce the overall CPU or I/O load on the database server, then identifying resource-intensive SQL involves the following steps:

1. Determine which period in the day you would like to examine; typically this is the application's peak processing time.
2. Gather OS and Oracle statistics over that period. The minimum of Oracle statistics gathered should be file I/O (V$FILESTAT), system statistics (V$SYSSTAT), and SQL statistics (V$SQLAREA or V$SQL, V$SQLTEXT and V$SQL_PLAN).

See Also: Chapter 21, "Using Statspack" for information on how to use Statspack to gather this data for you

3. Using the data collected in step two, identify the SQL statements using the most resources. A good way to identify candidate SQL statements is to query V$SQLAREA. V$SQLAREA contains resource usage information for all SQL statements in the shared pool. The data in V$SQLAREA should be ordered by resource usage. The most common resources are:
   - Buffer gets (V$SQLAREA.BUFFER_GETS, for high CPU using statements)
   - Disk reads (V$SQLAREA.DISK_READS, for high I/O statements)
   - Sorts (V$SQLAREA.SORTS, for many sorts)

   Note: One method to identify which SQL statements are creating the highest load is to compare the resources used by a SQL statement to the total amount of that resource used in the period. For BUFFER_GETS, divide each SQL statement's BUFFER_GETS by the total number of buffer gets during the period. This statistic is kept in the V$SYSSTAT table under the statistic `session logical reads`. Similarly, for DISK_READS, divide DISK_READS by the V$SYSSTAT statistic `physical reads`. This data is included in the Statspack report high load SQL section.

   See Also: Chapter 24, "Dynamic Performance Views for Tuning" for more information on V$SQLAREA and V$SQL

After the candidate SQL statements are identified, gather information to examine the statements and tune them.

Gathering Data on the SQL Identified

If you are most concerned with CPU, then examine the top SQL statements that performed the most BUFFER_GETS during that interval. Otherwise, start with the SQL statement that performed the most DISK_READS.

Information to Gather During Tuning

The tuning process begins by determining the structure of the underlying tables and indexes.

Information gathered includes the following:

1. The complete SQL text from V$SQLTEXT
2. The structure of the tables referenced in the SQL statement (usually by describing the table in SQL*Plus)
3. The definitions of any indexes (columns, column orderings), and whether the indexes are unique or nonunique
4. CBO statistics for the segments (including the number of rows each table, selectivity of the index columns), including the date when the segments were last analyzed
5. The definitions of any views referred to in the SQL statement
6. Repeat steps two and three above for any tables referenced in the view definitions found in step four
7. The optimizer plan for the SQL statement (either from EXPLAIN PLAN, V$SQL_PLAN, or the TKPROF output)
8. Any previous optimizer plans for that SQL statement

> **Note:** It is important to generate and review execution plans for all of the key SQL statements in your application. Doing so allows you to compare the optimizer execution plans of a SQL statement when the statement performed well to the plan when that the statement is not performing well. Having the comparison, along with information such as changes in data volumes, can assist in identifying the cause of performance degradation.

Overview of SQL Statement Tuning

This section describes ways you can improve SQL statement efficiency:

- Verifying Optimizer Statistics
- Reviewing the Execution Plan
- Restructuring the SQL Statements
- Restructuring the Indexes
- Modifying or Disabling Triggers and Constraints
- Restructuring the Data
- Maintaining Execution Plans Over Time
- Visiting Data as Few Times as Possible

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.

Verifying Optimizer Statistics

The CBO uses statistics gathered on tables and indexes when determining the optimal execution plan. If these statistics have not been gathered, or if the statistics are no longer representative of the data stored within the database, then the optimizer does not have sufficient information to generate the best plan.

Things to check:

- If you gather statistics for some tables in your database, then it is probably best to gather statistics for all tables. This is especially true if your application includes SQL statements that perform joins.
- If the optimizer statistics in the data dictionary are no longer representative of the data in the tables and indexes, then gather new statistics. One way to check whether the dictionary statistics are stale is to compare the cardinality from an `EXPLAIN PLAN` to the actual number of rows returned in each step of the `SQL_TRACE` (or `V$SQL_PLAN`). If these vary significantly, then it is likely that the statistics are stale.
- If there is significant data skew on predicate columns, then consider using histograms.

Reviewing the Execution Plan

When tuning (or writing) a SQL statement in an OLTP environment, the goal is to drive from the table that has the most selective filter. This means that there are fewer rows passed to the next step. If the next step is a join, then this means that fewer rows are joined. Check to see whether the access paths are optimal.

When examining the optimizer execution plan, look for the following:

- The plan is such that the driving table has the best filter
- The join order in each step means that the fewest number of rows are being returned to the next step
- The join method is appropriate for the number of rows being returned. For example, nested loop joins via indexes may not be optimal when many rows are being returned
- Views are used efficiently. Look at the SELECT list to see whether access to the view is necessary
- There are any unintentional Cartesian products (even with small tables)
- Each table is being accessed efficiently:

  Consider the predicates in the SQL statement and the number of rows in the table. Look for suspicious activity, such as a full table scans on tables with large number of rows, which have predicates in the where clause. Determine why an index is not used for such a selective predicate.

  A full table scan does not mean inefficiency. It might be more efficient to perform a full table scan on a small table, or to perform a full table scan to leverage a better join method (for example, hash_join) for the number of rows returned.

If any of the above are not optimal, then consider restructuring the SQL statement and / or the indexes available on the tables.

Restructuring the SQL Statements

Often, rewriting an inefficient SQL statement is easier than repairing it. If you understand the purpose of a given statement, then you might be able to quickly and easily write a new statement that meets the requirement.

Compose Predicates Using AND and =

To improve SQL efficiency, use equijoins whenever possible. Statements that perform equijoins on untransformed column values are the easiest to tune.

Avoid Transformed Columns in the WHERE Clause

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. 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, unless there is a function-based index defined that can be used.

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
```

Avoid the following kinds of complex expressions:

- col1 = NVL (:b1,col1)
- NVL (col1, -999) = ...
- TO_DATE(), TO_NUMBER(), and so on

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;
```

> **See Also:** Chapter 4, "Understanding Indexes and Clusters" for more information on function-based indexes

Write Separate SQL Statements for Specific Tasks

SQL is not a procedural language. Using one piece of SQL to do many different things usually results in a less-than-optimal result for each task. If you want SQL to accomplish different things, then write various statements, rather than writing one statement to do different things depending on the parameters you give it.

> **Note:** Oracle Forms and Reports are powerful development tools that 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. You can also invoke a server side PL/SQL package that 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 roundtrips and network traffic.

It is always better to write separate SQL statements for different tasks, but if you must use one SQL statement, then you can make a very complex statement slightly less complex by using the `UNION ALL` operator.

Optimization (determining the execution plan) takes place before the database knows what values will be substituted into the query. An execution plan cannot, 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 might 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, 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 of EXISTS versus IN for Subqueries

In certain circumstances, it is better to use IN rather than EXISTS. In general, if the selective predicate is in the subquery, then use IN. If the selective predicate is in the parent query, then use EXISTS.

Note: This discussion is most applicable in an OLTP environment, where the access paths either to the parent SQL or subquery are via indexed columns with high selectivity. In a DSS environment, there can be low selectivity in the parent SQL or subquery, and there might not be any indexes on the join columns. In a DSS environment, consider using semi-joins for the EXISTS case.

See Also:

- "How the CBO Executes Anti-Joins" on page 1-64
- "HASH_AJ, MERGE_AJ, and NL_AJ" on page 5-26 and "HASH_SJ, MERGE_SJ, and NL_SJ" on page 5-27
- *Oracle9i Data Warehousing Guide*

Sometimes, Oracle can rewrite a subquery when used with an IN clause to take advantage of selectivity specified in the subquery. This is most beneficial when the most selective filter appears in the subquery, and when there are indexes on the join columns.

Conversely, using EXISTS is beneficial when the most selective filter is in the parent query. This allows the selective predicates in the parent query to be applied before filtering the rows against the exists criteria.

Note: You should verify the CBO cost of the statement with the actual number of resources used (BUFFER_GETS, DISK_READS, CPU_TIME from V$SQL or V$SQLAREA). Situations such as data skew (without the use of histograms) can adversely affect the optimizer's estimated cost for an operation.

Below are two examples that demonstrate the benefits of IN and EXISTS. Both examples use the same schema with the following characteristics:

- There is a unique index on the employees.employee_id field.
- There is an index on the orders.customer_id field.
- There is an index on the employees.department_id field.
- The employees table has 27,000 rows.
- The orders table has 10,000 rows.
- The OE and HR schemas, which own the above segments, were both analyzed with COMPUTE.

Example 1: Using IN - Selective Filters in the Subquery This example demonstrates how rewriting a query to use IN can improve performance. This query identifies all employees who have placed orders on behalf of customer 144.

The SQL statement using EXISTS:

```
SELECT /* EXISTS example */
       e.employee_id
     , e.first_name
     , e.last_name
     , e.salary
  FROM employees e
 WHERE EXISTS (SELECT 1                                /* Note 1 */
                 FROM orders o
                WHERE e.employee_id = o.sales_rep_id  /* Note 2 */
                  AND o.customer_id = 144             /* Note 3 */
              );
```

Note:

- Note 1: This shows the line containing EXISTS.
- Note 2: This shows the line that makes the subquery a correlated subquery.
- Note 3: This shows the line where the correlated subqueries include the highly selective predicate customer_id = <number>.

Below is the execution plan (from V$SQL_PLAN) for the above statement. The plan requires a full table scan of the employees table, returning many rows. Each of these rows is then filtered against the orders table (via an index).

```
 ID OPERATION            OPTIONS         OBJECT_NAME            OPT       COST
---- -------------------- --------------- ---------------------- --- ----------
  0 SELECT STATEMENT                                            CHO
  1  FILTER
  2   TABLE ACCESS       FULL            EMPLOYEES              ANA        155
  3   TABLE ACCESS       BY INDEX ROWID  ORDERS                 ANA          3
  4    INDEX             RANGE SCAN      ORD_CUSTOMER_IX        ANA          1
```

Rewriting the statement using `IN` results in significantly fewer resources used.

The SQL statement using `IN`:

```
SELECT /* IN example */
       e.employee_id
     , e.first_name
     , e.last_name
     , e.salary
  FROM employees e
 WHERE e.employee_id IN (SELECT o.sales_rep_id       /* Note 4 */
                           FROM orders o
                          WHERE o.customer_id = 144  /* Note 3 */
                        );
```

Note:

- Note 3: This shows the line where the correlated subqueries include the highly selective predicate `customer_id` = <number>.
- Note 4: This indicates that an `IN` is being used. The subquery is no longer correlated, because the `IN` clause replaces the join in the subquery.

Below is the execution plan (from `V$SQL_PLAN`) for the above statement. The optimizer rewrites the subquery into a view, which is then joined via a unique index to the `employees` table. This results in a significantly better plan, because the view (that is, subquery) has a selective predicate, thus returning only a few `employee_ids`. These few `employee_ids` are then used to access the `employees` table via the unique index.

```
  ID OPERATION            OPTIONS         OBJECT_NAME           OPT       COST
---- -------------------- --------------- --------------------- --- ----------
   0 SELECT STATEMENT                                           CHO
   1  NESTED LOOPS                                                          5
   2   VIEW                                                                 3
   3    SORT               UNIQUE                                           3
   4     TABLE ACCESS      FULL            ORDERS                ANA        1
   5   TABLE ACCESS        BY INDEX ROWID  EMPLOYEES             ANA        1
   6    INDEX              UNIQUE SCAN     EMP_EMP_ID_PK         ANA
```

Example 2: Using EXISTS - Selective Predicate in the Parent This example demonstrates how rewriting a query to use `EXISTS` can improve performance. This query identifies all employees from department 80 who are sales reps who have placed orders.

SQL statement using `IN`:

```
SELECT /* IN example */
       e.employee_id
     , e.first_name
     , e.last_name
     , e.department_id
     , e.salary
  FROM employees   e
 WHERE e.department_id = 80                          /* Note 5 */
   AND e.job_id        = 'SA_REP'                    /* Note 6 */
   AND e.employee_id IN (SELECT o.sales_rep_id       /* Note 4 */
                           FROM orders o
                         );
```

Note:

- Note 4: This indicates that an `IN` is being used. The subquery is no longer correlated, because the `IN` clause replaces the join in the subquery.
- Note 5 and 6: These are the selective predicates in the parent SQL.

Below is the execution plan (from `V$SQL_PLAN`) for the above statement. The SQL statement was rewritten by the optimizer to use a view on the `orders` table, which requires sorting the data to return all unique `employee_ids` existing in the `orders` table. Because there is no predicate, many `employee_ids` are returned.

The large list of resulting `employee_ids` are then used to access the `employees` table via the unique index.

```
  ID OPERATION            OPTIONS         OBJECT_NAME            OPT       COST
---- -------------------- --------------- ---------------------- --- ----------
   0 SELECT STATEMENT                                            CHO
   1  NESTED LOOPS                                                          125
   2   VIEW                                                                 116
   3    SORT               UNIQUE                                           116
   4     TABLE ACCESS      FULL            ORDERS                 ANA        40
   5   TABLE ACCESS        BY INDEX ROWID  EMPLOYEES              ANA         1
   6    INDEX              UNIQUE SCAN     EMP_EMP_ID_PK          ANA
```

SQL statement using `EXISTS`:

```
SELECT /* EXISTS example */
       e.employee_id
     , e.first_name
     , e.last_name
     , e.salary
  FROM employees   e
 WHERE e.department_id = 80                          /* Note 5 */
   AND e.job_id        = 'SA_REP'                    /* Note 6 */
   AND EXISTS (SELECT 1                              /* Note 1 */
                 FROM orders o
                WHERE e.employee_id = o.sales_rep_id  /* Note 2 */
              );
```

Note:

- Note 1: This shows the line containing `EXISTS`.
- Note 2: This shows the line that makes the subquery a correlated subquery.
- Note 5 & 6:These are the selective predicates in the parent SQL.

Below is the execution plan (from `V$SQL_PLAN`) for the above statement. The cost of the plan is reduced by rewriting the SQL statement to use an `EXISTS`. This plan is more effective, because two indexes are used to satisfy the predicates in the parent query, thus returning only a few `employee_ids`. The `employee_ids` are then used to access the `orders` table via an index.

```
  ID OPERATION            OPTIONS         OBJECT_NAME            OPT       COST
---- -------------------- --------------- ---------------------- --- ----------
   0 SELECT STATEMENT                                            CHO
   1  FILTER
   2   TABLE ACCESS       BY INDEX ROWID  EMPLOYEES              ANA         98
   3    AND-EQUAL
   4     INDEX            RANGE SCAN      EMP_JOB_IX             ANA
   5     INDEX            RANGE SCAN      EMP_DEPARTMENT_IX      ANA
   6   INDEX              RANGE SCAN      ORD_SALES_REP_IX       ANA          8
```

Controlling the Access Path and Join Order with Hints

You can influence the optimizer's choices by setting the optimizer approach and goal, and by gathering representative 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. You can use hints in SQL statements to specify how the statement should be executed.

Hints, such as /*+FULL */ control access paths. For example:

```
SELECT /*+ FULL(e) */ e.ename
FROM emp e
WHERE e.job = 'CLERK';
```

> **See Also:** Chapter 1, "Introduction to the Optimizer" and Chapter 5, "Optimizer Hints"

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 100 AND 200
    AND b.bcol BETWEEN 10000 AND 20000
    AND c.ccol BETWEEN 10000 AND 20000
    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 previous example 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 is the one containing the filter condition that eliminates the highest percentage of the table. Thus, because the range of 100 to 200 is narrow compared with the range of `acol`, but the ranges of 10000 and 20000 are relatively large, `taba` is the driving table, all else being equal.

   With nested loop joins, the joins 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 do you use the indexes on the nonjoin conditions, except for the driving table. Thus, after `taba` is chosen as the driving table, use the indexes on `b.key1` and `c.key2` to drive into `tabb` and `tabc`, respectively.

2. 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`.

3. You can use the `ORDERED` or `STAR` hint to force the join order.

See Also: "Hints for Join Orders" on page 5-21

Use Caution When Managing Views

Be careful when joining views, when performing outer joins to views, and when reusing an existing view for a new purpose.

Use Caution When Joining Complex Views Joins to complex views are not recommended, particularly joins from one complex view to another. Often this results in the entire view being instantiated, and then the query is run against the view data.

For example:

```
View:
List of employees and departments

CREATE OR REPLACE VIEW emp_dept
AS
SELECT d.department_id
     , d.department_name
     , d.location_id
     , e.employee_id
     , e.last_name
     , e.first_name
     , e.salary
     , e.job_id
FROM  departments d
     ,employees e
WHERE e.department_id (+) = d.department_id
/

Query:
Finds employees in a given state

SELECT v.last_name, v.first_name, l.state_province
FROM locations l, emp_dept v
WHERE l.state_province = 'California'
AND   v.location_id = l.location_id (+)
/

Plan:  Note, the emp_dept view is instantiated
Plan Table
--------------------------------------------------------------------------------
| Operation                 |  Name    |  Rows | Bytes|  Cost  | Pstart| Pstop |
--------------------------------------------------------------------------------
| SELECT STATEMENT          |          |       |      |        |       |       |
|  FILTER                   |          |       |      |        |       |       |
|   NESTED LOOPS OUTER      |          |       |      |        |       |       |
|    VIEW                   |EMP_DEPT  |       |      |        |       |       |
|     NESTED LOOPS OUTER    |          |       |      |        |       |       |
|      TABLE ACCESS FULL    |DEPARTMEN |       |      |        |       |       |
|      TABLE ACCESS BY INDEX|EMPLOYEES |       |      |        |       |       |
|       INDEX RANGE SCAN    |EMP_DEPAR |       |      |        |       |       |
|    TABLE ACCESS BY INDEX R|LOCATIONS |       |      |        |       |       |
|     INDEX UNIQUE SCAN     |LOC_ID_PK |       |      |        |       |       |
--------------------------------------------------------------------------------
```

Do Not Recycle Views Beware of writing a view for one purpose and then using it for other purposes to which it might be ill-suited. Querying from a view requires all tables from the view to be accessed for the data to be returned. Before reusing a view, determine whether all tables in the view need to be accessed to return the data. If not, then do not use the view. Instead, use the base table(s), or if necessary, define a new view. The goal is to refer to the minimum number of tables and views necessary to return the required data.

Consider the following example:

```
SELECT dname
FROM dx
WHERE deptno=10;
```

The entire view is first instantiated by performing a join of the `emp` and `dept` tables and then aggregating the data. However, you can obtain `dname` and `deptno` directly from the `dept` table. It is inefficient to obtain this information by querying the `dx` view (which was declared in the earlier example).

Use Caution When Unnesting Subqueries Subquery unnesting 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.

> **See Also:** *Oracle9i Data Warehousing Guide* for an explanation of the dangers with subquery unnesting

Use Caution When Performing Outer Joins to Views An outer join to a multitable view can be problematic. You cannot drive from an outer join column without first instantiating the view.

An outer join within a view is also problematic, because the performance implications of the outer join are not visible.

Store Intermediate Results

Intermediate, or staging, tables are quite common in relational database systems, because they temporarily store some intermediate results. In many applications they are useful, but Oracle requires additional resources to create them. Always consider whether the benefit they could bring is more than the cost to create them. Avoid staging tables when the information is not reused multiple times.

Some additional considerations:

- Storing intermediate results in staging tables could improve application performance. In general, whenever an intermediate result is usable by multiple

following queries, it is worthwhile to store it in a staging table. The benefit of not retrieving data multiple times with a complex statement already at the second usage of the intermediate result is better than the cost to materialize it.

- Long and complex queries are hard to understand and optimize. Staging tables can break a complicated SQL statement into several smaller statements, and then store the result of each step.
- Consider using materialized views. These are precomputed tables comprising aggregated and / or joined data from fact and possibly dimension tables.

  See Also: *Oracle9i Data Warehousing Guide* for detailed information on using materialized views

Restructuring the Indexes

Often, there is a beneficial impact on performance by restructuring indexes. This can involve the following:

- Remove nonselective indexes to speed the DML.
- Index performance-critical access paths.
- Consider reordering columns in existing concatenated indexes.
- Add columns to the index to improve selectivity.

Do not use indexes as a panacea. Application developers sometimes think that performance will improve if they create 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.

Modifying or Disabling Triggers and Constraints

Using triggers consumes system resources. If you use too many triggers, then you can find that performance is adversely affected and you might 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.
- Review your data design. Change the design of your system if it can improve performance.
- Consider partitioning, if appropriate.

Maintaining Execution Plans Over Time

You can maintain the existing execution plan of SQL statements over time either using stored statistics or stored SQL execution plans. Storing optimizer statistics for tables will apply to all SQL statements that refer to those tables. Storing an execution plan (that is, plan stability) maintains the plan for a single SQL statement. If both statistics and a stored plan are available for a SQL statement, then the optimizer uses the stored plan.

See Also:

- Chapter 3, "Gathering Optimizer Statistics"
- Chapter 7, "Using Plan Stability"

Visiting Data as Few Times as Possible

Applications should try to access each row only once. This reduces network traffic and reduces database load. Consider doing the following:

- Combine Multiples Scans with CASE Statements
- Use DML with RETURNING Clause
- Modify All the Data you Need in One Statement

Combine Multiples Scans with CASE Statements

Often, it is necessary to calculate different aggregates on various sets of tables. Usually, this is done with multiple scans on the table, but it is easy to calculate all the aggregates with one single scan. Eliminating n-1 scans can greatly improve performance.

Combining multiple scans into one scan can be done by moving the WHERE condition of each scan into a CASE statement, which filters the data for the aggregation. For each aggregation, there could be another column that retrieves the data.

The following example asks for the count of all employees who earn less then 2000, between 2000 and 4000, and more than 4000 each month. This can be done with three separate queries:

```
SELECT COUNT (*)
FROM employees
WHERE salary<2000;

SELECT COUNT (*)
FROM employees
WHERE salary BETWEEN 2000 AND 4000;

SELECT COUNT (*)
FROM employees
WHERE salary>4000;
```

However, it is more efficient to run the entire query in a single statement. Each number is calculated as one column. The count uses a filter with the CASE statement to count only the rows where the condition is valid. For example:

```
SELECT COUNT (CASE WHEN salary < 2000
                   THEN 1 ELSE null END) count1,
       COUNT (CASE WHEN salary BETWEEN 2001 AND 4000
                   THEN 1 ELSE null END) count2,
       COUNT (CASE WHEN salary > 4000
                   THEN 1 ELSE null END) count3
  FROM employees;
```

This is a very simple example. The ranges could be overlapping, the functions for the aggregates could be different, and so on.

Use DML with RETURNING Clause

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:** *Oracle9i SQL Reference* for syntax on the INSERT, UPDATE, and DELETE statements

Modify All the Data you Need in One Statement

When possible, use array processing. This means that an array of bind variable values is passed to Oracle for repeated execution. This is appropriate for iterative processes in which multiple rows of a set are subject to the same operation.

For example:

```
BEGIN
 FOR pos_rec IN (SELECT *
   FROM order_positions
   WHERE order_id = :id) LOOP
      DELETE FROM order_positions
      WHERE order_id = pos_rec.order_id AND
        order_position = pos_rec.order_position;
 END LOOP;
 DELETE FROM orders
 WHERE order_id = :id;
END;
```

Alternatively, you could define a cascading constraint on `orders`. In the previous example, one `SELECT` and *n* `DELETE`s are executed. When a user issues the `DELETE` on `orders` `DELETE FROM orders WHERE order_id = :id`, the database automatically deletes the positions with a single `DELETE` statement.

> **See Also:** *Oracle9i Database Administrator's Guide* or *Oracle9i Heterogeneous Services* for information on tuning distributed queries

7

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
- Using Plan Stability with the Cost-Based Optimizer

Using Plan Stability to Preserve Execution Plans

Plan stability prevents certain database environment changes from affecting the performance characteristics of 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 an application.

Plan stability preserves execution plans in *stored outlines*. Oracle can create a public or private 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. You can group outlines into categories and control which category of outlines Oracle uses to simplify outline administration and deployment.

The plans Oracle maintains in stored outlines remain consistent despite changes to a system's configuration or statistics. Using stored outlines also stabilizes the generated execution plan if the optimizer changes in subsequent Oracle releases. 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 customers access the same execution plans.

Using Hints with Plan Stability

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.

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 applications with bind variables.

> **See Also:** Oracle can force similar statements to share SQL by replacing literals with system-generated bind variables. This works with plan stability if the outline was generated using the CREATE_STORED_OUTLINES parameter, not the CREATE OUTLINE statement. Also, the outline must have been created with the CURSOR_SHARING parameter set to SIMILAR or FORCE, and the parameter must also set to SIMILAR or FORCE when attempting to use the outline. See Chapter 14, "Memory Configuration and Use" for more information.

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 can 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: Oracle creates the USER_OUTLINES and USER_OUTLINE_HINTS views in the SYS tablespace based on data in the OL$ and OL$HINTS tables, respectively. Direct manipulation of the OL$, OL$HINTS, and OL$NODES tables 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.

Storing Outlines

Oracle stores outline data in the OL$, OL$HINTS, and OL$NODES tables. 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

Using Supplied Packages to Manage Stored Outlines

The DBMS_OUTLN and DBMS_OUTLN_EDIT package provides procedures used for managing stored outlines and their outline categories.

Users need the EXECUTE_CATALOG_ROLE role to execute DBMS_OUTLN, but public has execute privileges on DBMS_OUTLN_EDIT. The DBMS_OUTLN_EDIT package is an invoker's rights package.

> **See Also:** *Oracle9i Supplied PL/SQL Packages Reference* for detailed information on using DBMS_OUTLN and DBMS_OUTLN_EDIT procedures

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. You can create stored outlines for specific statements using the CREATE OUTLINE statement.

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 the database after you run the application.

Also, the default system tablespace can become exhausted if the CREATE_STORED_OUTLINES parameter is enabled and the running application has an abundance of literal SQL statements. If this happens, use the DBMS_OUTLN.DROP_UNUSED procedure to remove those literal SQL outlines.

The CREATE_EDIT_TABLES procedure in the DBMS_OUTLN_EDIT package creates tables in the invoker's schema. This is necessary for editing private outlines. This is callable by anyone with EXECUTE privilege on DBMS_OUTLN_EDIT.

See Also:

- *Oracle9i SQL Reference* for more information on the CREATE OUTLINE statement
- *Oracle9i Supplied PL/SQL Packages Reference* for more information on the DBMS_OUTLN and DBMS_OUTLN_EDIT packages
- "Using Outlines to Move to the Cost-Based Optimizer" on page 7-11 for information on moving from the rule-based optimizer to the cost-based optimizer
- *Database Tuning with the Oracle Tuning Pack* for information on using the Outline Management and Outline Editor tools, which let you create, edit, delete, and manage stored outlines with an easy-to-use graphical interface

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. 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`, `USE_STORED_OUTLINES`, and `USE_PRIVATE_OUTLINES` parameters are system- or session-specific. They are not initialization parameters. For more information on these parameters, see the *Oracle9i SQL Reference.*

Using and Editing Stored Outlines

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.

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 reset 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.

The `USE_PRIVATE_OUTLINES` parameter lets you control the use of private outlines. A private outline is an outline seen only in the current session and whose data resides in the current parsing schema. Any changes made to such an outline are not seen by any other session on the system, and applying a private outline to the compilation of a statement can only be done in the current session with the

USE_PRIVATE_OUTLINES parameter. Only when you explicitly choose to save your edits back to the public area are they seen by the rest of the users.

While the optimizer usually chooses optimal plans for queries, there are times when users know things about the execution environment that are inconsistent with the heuristics that the optimizer follows. By editing outlines directly, you can tune the SQL query without having to alter the application.

When a private outline is created, an error is returned if the prerequisite outline tables to hold the outline data do not exist in the local schema. These tables can be created using the DBMS_OUTLN_EDIT.CREATE_EDIT_TABLES procedure. You can also use the UTLEDITOL.SQL script.

When the USE_PRIVATE_OUTLINES parameter is enabled and an outlined SQL statement is issued, the optimizer retrieves the outline from the session private area rather than the public area used when USE_STORED_OUTLINES is enabled. If no outline exists in the session private area, then the optimizer will not use an outline to compile the statement.

Any CREATE OUTLINE statement requires the CREATE ANY OUTLINE privilege. Specification of the FROM clause also requires the SELECT privilege. This privilege should be granted only to those users who would have the authority to view SQL text and hint text associated with the outlined statements. This role is required for the CREATE OUTLINE FROM command unless the issuer of the command is also the owner of the outline.

When you begin an editing session, USE_PRIVATE_OUTLINES should be set to the category to which the outline being edited belongs. When you are done editing, this parameter should be set to `false` to restore the session to normal outline lookup as per the USE_STORED_OUTLINES parameter.

Example of Editing Outlines

Assume that you want to edit the outline `ol1`. The steps are as follows:

1. Connect to a schema from which the outlined statement can be executed, and ensure that the CREATE ANY OUTLINE and SELECT privileges have been granted.
2. Create outline editing tables locally with the DBMS_OUTLN_EDIT.CREATE_EDIT_TABLES procedure.
3. Clone the outline being edited to the private area using the following:

```
CREATE PRIVATE OUTLINE p_ol1 FROM ol1;
```

4. Edit the outline, either with the Outline Editor in Enterprise Manager or manually by querying the local OL$HINTS tables and performing DML against the appropriate hint tuples. DBMS_OUTLN_EDIT.CHANGE_JOIN_POS is available for changing join order.
5. If manually editing the outline, then resynchronize the stored outline definition using the following so-called identity statement:

   ```
   CREATE PRIVATE OUTLINE p_ol1 FROM PRIVATE p_ol1;
   ```

   You can also use DBMS_OUTLN_EDIT.REFRESH_PRIVATE_OUTLINE or ALTER SYSTEM FLUSH SHARED_POOL to accomplish this.
6. Test the edits. Set USE_PRIVATE_OUTLINES=true, and issue the outline statement or run EXPLAIN PLAN on the statement.
7. If you want to preserve these edits for public use, then publicize the edits with the following statement.

   ```
   CREATE OR REPLACE OUTLINE ol1 FROM PRIVATE p_ol1;
   ```

8. Disable private outline usage by setting the following:

   ```
   USE_PRIVATE_OUTLINES=false
   ```

See Also:

- *Oracle9i SQL Reference* for SQL syntax
- *Database Tuning with the Oracle Tuning Pack* for more information on the GUI tool for editing outlines
- *Oracle9i Supplied PL/SQL Packages Reference* for more information on the DBMS_OUTLN and DBMS_OUTLN_EDIT packages

How to Tell if an Outline is Being Used

You can 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. The OUTLINE_SID column tells you if this particular cursor is using a public outline (value is 0) or a private outline (session's SID of the corresponding session using it).

For example:

```
SELECT OUTLINE_CATEGORY, OUTLINE_SID
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';
```

Note: If necessary, you can use the procedure to move outline tables from one tablespace to another as described in "Moving Outline Tables" on page 7-9.

Moving Outline Tables

Oracle creates the USER_OUTLINES and USER_OUTLINE_HINTS views based on data in the OL$ and OL$HINTS tables, respectively. Oracle creates these tables, and also the OL$NODES table, in the SYS tablespace using a schema called OUTLN. If

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. The default system tablespace could become exhausted if the CREATE_STORED_OUTLINES parameter is on and if the running application has many literal SQL statements. If this happens, then use the DBMS_OUTLN.DROP_UNUSED procedure to remove those literal SQL outlines.

   See Also: *Oracle9i Supplied PL/SQL Packages Reference* for detailed information on using the DBMS_OUTLN package

2. Export the OL$, OL$HINTS, and OL$NODES tables:

   ```
   EXP OUTLN/OUTLN FILE = exp_file TABLES = 'OL$' 'OL$HINTS' 'OL$NODES'
   ```

3. Remove the previous OL$, OL$HINTS, and OL$NODES tables:

   ```
   CONNECT OUTLN/outln_password;
   DROP TABLE OL$;
   CONNECT OUTLN/outln_password;
   DROP TABLE OL$HINTS;
   CONNECT OUTLN/outln_password;
   DROP TABLE OL$NODES;
   ```

4. 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;
   ```

5. Enter the following statement:

   ```
   ALTER USER OUTLN DEFALUT TABLESPACE outln_ts;
   ```

6. Import the OL$, OL$HINTS, and OL$NODES tables:

   ```
   IMP OUTLN/outln_password
   FILE=exp_file TABLES = 'OL$' 'OL$HINTS' 'OL$NODES'
   ```

   The IMPORT statement re-creates the OL$, OL$HINTS, and OL$NODES tables in the schema named OUTLN, but the schema now resides in a new tablespace called OUTLN_TS.

Note: If Oracle8*i* outlines are imported into an Oracle9*i* database, then the DBMS_OUTLN.UPDATE_SIGNATURES procedure must be run. This updates the signatures of all outlines on the system so that they are compatible with Oracle9*i* semantics. If this step is not done, then no outlines from the Oracle8*i* database are used.

Using Plan Stability with 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 a system's targeted execution plans with satisfactory performance while also taking advantage of new cost-based optimizer features for the rest of the SQL statements.

Topics covered in this section are:

- Using Outlines to Move to the Cost-Based Optimizer
- Upgrading and the Cost-Based Optimizer

Using Outlines to Move to the Cost-Based Optimizer

If an application was developed using the rule-based optimizer, then a considerable amount of effort might 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 the 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 the following 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.

Upgrading 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, you might have applications that perform so well that 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 running 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 can 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' 'OL$NODES'
QUERY='WHERE CATEGORY="problemcat"'
```

8

Using the Rule-Based Optimizer

This chapter discusses Oracle's rule-based optimizer (RBO). In general, always use the cost-based approach. The rule-based approach is available for backward compatibility.

See Also: Chapter 1, "Introduction to the Optimizer"

This chapter contains the following sections:

- Overview of the Rule-Based Optimizer (RBO)
- Understanding Access Paths for the RBO
- Transforming and Optimizing Statements with the RBO

Overview of the Rule-Based Optimizer (RBO)

Although Oracle supports the rule-based optimizer, you should design new applications to use the cost-based optimizer (CBO). 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 the SQL statements carefully based on the rules of the optimizer, then you might 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 SQL statements, then SQL statements use the RBO. You can use the RBO to access both relational data and object types. If `OPTIMIZER_MODE=FIRST_ROWS`, `FIRST_ROWS_`*n*, or `ALL_ROWS` and no statistics exist, then the CBO uses default statistics. Migrate 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 data without using the cost-based approach.

> **See Also:** Chapter 3, "Gathering Optimizer Statistics" for an explanation of how to gather statistics

Understanding 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:

RBO Path 1: Single Row by Rowid

RBO Path 2: Single Row by Cluster Join

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

RBO Path 4: Single Row by Unique or Primary Key

RBO Path 5: Clustered Join

RBO Path 6: Hash Cluster Key

RBO Path 7: Indexed Cluster Key

RBO Path 8: Composite Index

RBO Path 9: Single-Column Indexes

RBO Path 10: Bounded Range Search on Indexed Columns

RBO Path 11: Unbounded Range Search on Indexed Columns

RBO Path 12: Sort Merge Join

RBO Path 13: MAX or MIN of Indexed Column

RBO Path 14: ORDER BY on Indexed Column

RBO 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.

RBO 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.

For 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
```

RBO 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 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: "Nested Loop Outer Joins" on page 1-48

For 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.

RBO 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.

For 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
```

RBO 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.

For 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.

RBO 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: "Nested Loop Outer Joins" on page 1-48

For 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
```

RBO 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.

For 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
```

RBO 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.

For 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.

RBO 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.

For 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.

RBO 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.

For example:

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.

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.

RBO 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.

For example:

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`.

In the following statement, there is an index on the `ename` column of the `emp` table:

```
SELECT *
    FROM emp
    WHERE ename LIKE 'S%';
```

RBO 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.

For example:

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
```

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.

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.

RBO 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:** "Understanding Joins" on page 1-42 for information on these operations

For 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
```

RBO 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 full 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.

For 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:

```
0      SELECT STATEMENT Optimizer=CHOOSE
1    0  SORT (AGGREGATE)
2    1    INDEX (FULL SCAN (MIN/MAX)) OF 'SAL_INDEX' (NON-UNIQUE)
```

RBO 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 initialization 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.

For 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.

RBO 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.

For 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
```

Choosing Execution Plans for Joins with the RBO

Note: 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.

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:

- 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 "Understanding Access Paths for the RBO" on page 8-2. The optimizer repeats this step to fill each subsequent position in the join order.
- 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:

   - The optimizer chooses the execution plan with the fewest nested-loops operations in which the inner table is accessed with a full table scan.
   - If there is a tie, then the optimizer chooses the execution plan with the fewest sort-merge operations.
   - 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.
- 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.

Transforming and Optimizing Statements with the RBO

SQL is a very flexible query language; often, there are 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 with the RBO
- Using Alternative SQL Syntax

Transforming ORs into Compound Queries with the RBO

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 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.

See Also: "Understanding Access Paths for the RBO" on page 8-2 for information on access paths and how indexes make them available

For 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;
```

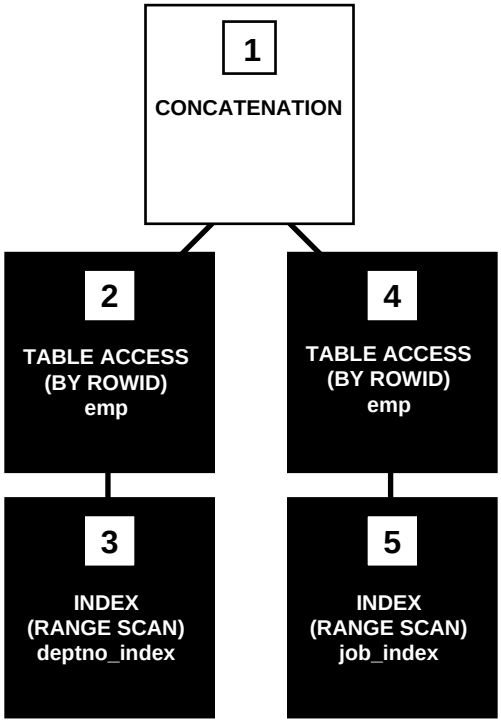
If there are indexes on both the job and deptno columns, then the optimizer might transform this query into the following equivalent query:

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

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 8–1.

Figure 8–1 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.

For 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 previous query would result in the following compound query:

```
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.

Using Alternative SQL Syntax

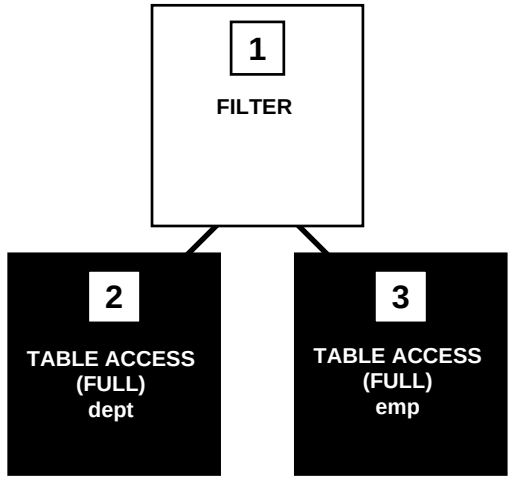
Because SQL is a flexible language, more than one SQL statement can meet the needs of an application. Although two SQL statements can produce the same result, Oracle might process one faster than the other. You can use the results of the EXPLAIN PLAN statement to compare the execution plans and costs 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 8–2 Execution Plan with Two Full Table Scans

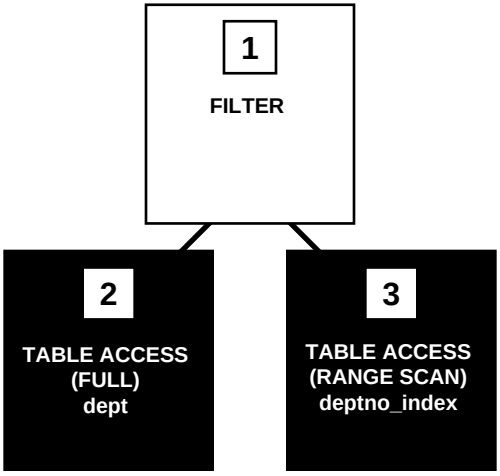


Step 3 of the output indicates that Oracle executes this statement by performing a full table scan of the `emp` table despite the index on the `deptno` column. This full table scan can be a time-consuming operation. Oracle does not use the index, because the subquery that searches the `emp` table does 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 8–3 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 an applications that use the `NOT IN` operator, as the first query in this example does, then consider rewriting them so that they use the `NOT EXISTS` operator. This allows such statements to use an index if one exists.

Note: Alternative SQL syntax is effective only with the rule-based optimizer.

Part II

SQL-Related Performance Tools

Oracle's SQL-related performance tools let you examine the execution plan for a SQL statement, and thereby determine whether the statement can be better optimized. You can get the execution plan from the EXPLAIN PLAN SQL statement, from querying V$SQL_PLAN, or from SQL trace.

In the development phase, use EXPLAIN PLAN to determine a good access plan, and then verify that it is the optimal plan through volume data testing. When evaluating a plan, examine the statement's actual resource consumption using V$SQLAREA with V$SQL_PLAN, Oracle Trace, or the SQL trace facility and TKPROF. The information in the V$SQL_PLAN view is very similar to the output of an EXPLAIN PLAN statement. However, EXPLAIN PLAN shows a theoretical plan that can be used if the statement were to be executed, whereas V$SQL_PLAN contains the actual plan used. Hence, querying V$SQLAREA in conjunction with V$SQL_PLAN provides similar results to using SQL Trace with TKPROF.

See Also: Chapter 24, "Dynamic Performance Views for Tuning"

The autotrace tool lets you automatically get a report on the execution path used by the SQL optimizer and the statement execution statistics. It is useful for monitoring and tuning the performance of these statements. Oracle Trace is a GUI, event-driven data collection product, which the Oracle server uses to collect performance and resource utilization data.

The chapters in this part are:

- Chapter 9, "Using EXPLAIN PLAN"
- Chapter 10, "Using SQL Trace and TKPROF"
- Chapter 11, "Using Autotrace in SQL*Plus"
- Chapter 12, "Using Oracle Trace"

9

Using EXPLAIN PLAN

This chapter introduces execution plans, describes the SQL statement EXPLAIN PLAN, and explains how to interpret its output. This chapter also provides procedures for managing outlines to control application performance characteristics.

This chapter contains the following sections:

- Understanding EXPLAIN PLAN
- Creating the PLAN_TABLE Output Table
- Running EXPLAIN PLAN
- Displaying PLAN_TABLE Output
- Reading EXPLAIN PLAN Output
- Viewing Bitmap Indexes with EXPLAIN PLAN
- Viewing Partitioned Objects with EXPLAIN PLAN
- EXPLAIN PLAN Restrictions
- PLAN_TABLE Columns

See Also: *Oracle9i SQL Reference* for the syntax of EXPLAIN PLAN

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 run the statement.

The row source tree is the core of the execution plan. It shows the following information:

- 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
- Data operations like filter, sort, or aggregation

In addition to the row source tree, the plan table contains information about the following:

- Optimization, such as the cost and cardinality of each operation
- Partitioning, such as the set of accessed partitions
- Parallel execution, such as the distribution method of join inputs

The EXPLAIN PLAN results let you determine whether the optimizer selects a particular execution plan (for example, nested loops join). It also helps you to understand the optimizer decisions (for example, why the optimizer chose a nested loops join instead of a hash join), and explains the performance of a query.

Note: Oracle Performance Manager charts and Oracle SQL Analyze can automatically create and display explain plans for you. For more information on using explain plans, see the *Database Tuning with the Oracle Tuning Pack* manual.

How Execution Plans Can Change

With the cost-based optimizer, execution plans can and do change as the underlying costs change. EXPLAIN PLAN output shows how Oracle runs the SQL statement when the statement was explained. This can differ from the plan during actual execution for a SQL statement, because of differences in the execution environment and explain plan environment.

Execution plans can differ due to the following:

- Different Schemas
- Different Costs

Different Schemas

- The execution and explain plan happen on different databases.
- The user explaining the statement is different from the user running the statement. Two users might be pointing to different objects in the same database, resulting in different execution plans.
- Schema changes (usually changes in indexes) between the two operations.

Different Costs

Even if the schemas are the same, the optimizer can choose different execution plans if the costs are different. Some factors that affect the costs include the following:

- Data volume and statistics
- Bind variable types
- Initialization parameters (changed globally or at session level)

Looking Beyond Execution Plans

The execution plan operation alone cannot differentiate between well-tuned statements and those that perform poorly. For example, an EXPLAIN PLAN output that shows that a statement uses an index does not necessarily mean that the statement runs efficiently. Sometimes indexes can be extremely inefficient. In this case, you should examine the following:

- The columns of the index being used
- Their selectivity (fraction of table being accessed)

It is best to use EXPLAIN PLAN to determine an access plan, and then later prove that it is the optimal plan through testing. When evaluating a plan, examine the statement's actual resource consumption. Use Oracle Trace or the SQL trace facility and TKPROF to examine individual SQL statement performance.

> **See Also:** Chapter 10, "Using SQL Trace and TKPROF" for information on TKPROF interpretation

Creating the PLAN_TABLE Output Table

Before issuing an EXPLAIN PLAN statement, you must have a table to hold it's output.

Use 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, it is located under $ORACLE_HOME/rdbms/admin on Unix. PLAN_TABLE is the default table into which the EXPLAIN PLAN statement inserts rows describing execution plans.

Oracle Corporation recommends that you drop and rebuild the PLAN_TABLE table after upgrading the version of the database. This is because the columns might change, which can cause scripts (or TKPROF, if you are specifying the table) to fail.

If you want an output table with a different name, then create PLAN_TABLE and rename it with the RENAME SQL statement.

Running EXPLAIN PLAN

To explain a SQL statement, enter the following:

```
EXPLAIN PLAN FOR
<<SQL Statement>>
```

This explains the plan into the PLAN_TABLE table. You can then select the execution plan from PLAN_TABLE. This is useful if you do not have any other plans in PLAN_TABLE, or if you only want to look at the last statement.

For example:

```
EXPLAIN PLAN FOR
SELECT name FROM emp;
```

Identifying Statements for EXPLAIN PLAN

With multiple statements, you can specify a statement identifier and use that to identify your specific execution plan. Before using a statement ID, remove any existing rows for that statement ID.

For example:

```
EXPLAIN PLAN
    SET STATEMENT_ID = <<identifier>> FOR
<<SQL Statement>>
```

Specifically:

```
EXPLAIN PLAN
    SET STATEMENT_ID = 'bad1' FOR
SELECT name FROM emp;
```

Specifying Different Tables for EXPLAIN PLAN

You can specify the INTO clause to specify a different table.

For example:

```
EXPLAIN PLAN
   INTO <<different table>>
 FOR <<SQL Statement>>
```

Specifically:

```
EXPLAIN PLAN
   INTO my_plan_table
    FOR
SELECT name FROM emp;
```

Or:

```
EXPLAIN PLAN
   INTO my_plan_table
    SET STATEMENT_ID = 'bad1' FOR
SELECT name FROM emp;
```

> **See Also:** *Oracle9i Database Reference* for a complete description of EXPLAIN PLAN syntax.

Displaying PLAN_TABLE Output

After you have explained the plan, use the two scripts provided by Oracle to display the most recent plan table output:

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

If you have specified a statement identifier, then you can write your own script to query the PLAN_TABLE.

For example:

- Start with ID = 0 and given STATEMENT_ID.
- Use the CONNECT BY clause to walk the tree from parent to child, the join keys being STATEMENT_ID = PRIOR STATEMENT_ID and PARENT_ID = PRIOR ID.
- Use the pseudo-column LEVEL (associated with CONNECT BY) to indent the children.

```
SELECT cardinality "Rows",
       lpad(' ',level-1)||operation||' '||
       options||' '||object_name "Plan"
  FROM PLAN_TABLE

CONNECT BY prior id = parent_id
        AND prior statement_id = statement_id
  START WITH id = 0
        AND statement_id = 'bad1'
  ORDER BY id;

   Rows Plan
------- ---------------------------------------
        SELECT STATEMENT
         TABLE ACCESS FULL EMP
```

The NULL in the Rows column indicates that the optimizer does not have any statistics on the table. Analyzing the table shows the following:

```
   Rows Plan
------- ---------------------------------------
  16957 SELECT STATEMENT
  16957  TABLE ACCESS FULL EMP
```

You can also select the COST. This is useful for comparing execution plans or for understanding why the optimizer chooses one execution plan over another.

Reading EXPLAIN PLAN Output

This section uses progressively complex examples to illustrate execution plans.

See Also: Appendix A, "Schemas Used in Performance Examples"

The following statement is used to display the execution plan:

```
SELECT lpad(' ',level-1)||operation||' '||options||' '||
        object_name "Plan"
   FROM plan_table
CONNECT BY prior id = parent_id
        AND prior statement_id = statement_id
  START WITH id = 0 AND statement_id = '&1'
  ORDER BY id;
```

EXPLAIN PLAN Example 1

```
EXPLAIN PLAN SET statement_id = 'example1' FOR
SELECT full_name FROM per_all_people_f
 WHERE UPPER(full_name) LIKE 'Pe%' ;

Plan
----------------------------------------------
SELECT STATEMENT
 TABLE ACCESS FULL PER_ALL_PEOPLE_F
```

This plan shows execution of a SELECT statement. The table PER_ALL_PEOPLE_F is accessed using a full table scan.

- Every row in the table PER_ALL_PEOPLE_F is accessed, and the WHERE clause criteria is evaluated for every row.
- The SELECT statement returns the rows meeting the WHERE clause criteria.

EXPLAIN PLAN Example 2

```
EXPLAIN PLAN SET statement_id = 'example2' FOR
SELECT full_name FROM per_all_people_f
 WHERE full_name LIKE 'Pe%' ;

Plan
----------------------------------------------
SELECT STATEMENT
 TABLE ACCESS BY INDEX ROWID PER_ALL_PEOPLE_F
  INDEX RANGE SCAN PER_PEOPLE_F_N54
```

This plan shows execution of a SELECT statement.

- Index PER_PEOPLE_F_N54 is used in a range scan operation.
- The table PER_ALL_PEOPLE_F is accessed via ROWID. ROWIDs are obtained from the index in the previous step for keys that meet the WHERE clause criteria. When the table is accessed, any additional WHERE clause conditions that could

not be evaluated during the range scan (because the column is present in the table and not in the index) are also evaluated.

- The SELECT statement returns rows satisfying the WHERE clause conditions (evaluated in previous steps).

EXPLAIN PLAN Example 3

```
EXPLAIN PLAN SET statement_id = 'example3' FOR
SELECT segment1, segment2, description, inventory_item_id
  FROM mtl_system_items msi
 WHERE segment1 = :b1
   AND segment2 LIKE '%-BOM'
AND NVL(end_date_active,sysdate+1) > SYSDATE ;

Plan
-----------------------------------------------
SELECT STATEMENT
 TABLE ACCESS BY INDEX ROWID MTL_SYSTEM_ITEMS
  INDEX RANGE SCAN MTL_SYSTEM_ITEMS_N8
```

This plan shows execution of a SELECT statement.

- Index MTL_SYSTEM_ITEMS_N8 is used in a range scan operation. This is an index on (SEGMENT1, SEGMENT2, SEGMENT3). The range scan happens using the following condition:

  ```
  segment1 = :b1
  ```

  The rows that come out of this step satisfy all the WHERE clause criteria that can be evaluated with the index columns. Therefore, the following condition is also evaluated at this stage:

  ```
  segment2 LIKE '%-BOM'
  ```

- The table PER_ALL_PEOPLE_F is accessed via ROWIDs obtained from the index in the previous step. When the table is accessed, any additional WHERE clause conditions that could not be evaluated during the range scan (because the column is present in the table and not in the index) are also evaluated. Therefore, the following condition is evaluated at this stage:

  ```
  NVL(end_date_active,sysdate+1) > SYSDATE
  ```

- The SELECT statement returns rows satisfying the WHERE clause conditions (evaluated in previous steps).

EXPLAIN PLAN Example 4

```
EXPLAIN PLAN SET statement_id = 'example4' FOR
SELECT h.order_number, l.revenue_amount, l.ordered_quantity
  FROM so_headers_all h, so_lines_all l
 WHERE h.customer_id = :b1
   AND h.date_ordered > SYSDATE-30
   AND l.header_id = h.header_id ;

Plan
-------------------------------------------------
SELECT STATEMENT
 NESTED LOOPS
  TABLE ACCESS BY INDEX ROWID SO_HEADERS_ALL
   INDEX RANGE SCAN SO_HEADERS_N1
  TABLE ACCESS BY INDEX ROWID SO_LINES_ALL
   INDEX RANGE SCAN SO_LINES_N1
```

This plan shows execution of a SELECT statement.

- Index so_headers_n1 is used in a range scan operation. This is an index on customer_id. The range scan happens using the following condition:

  ```
  customer_id = :b1
  ```

- The table so_headers_all is accessed via ROWIDs obtained from the index in the previous step. When the table is accessed, any additional WHERE clause conditions that could not be evaluated during the range scan (because the column is present in the table and not in the index) are also evaluated. Therefore, the following condition is evaluated at this stage:

  ```
  h.date_ordered > sysdate-30
  ```

- For every row from so_headers_all satisfying the WHERE clause conditions, a range scan is run on so_lines_n1 using the following condition:

  ```
  l.header_id = h.header_id
  ```

- The table so_lines_all is accessed via ROWIDs obtained from the index in the previous step. When the table is accessed, any additional WHERE clause conditions that could not be evaluated during the range scan (because the column is present in the table and not in the index) are also evaluated. There are no additional conditions to evaluate here.
- The SELECT statement returns rows satisfying the WHERE clause conditions (evaluated in previous steps).

Viewing Bitmap Indexes with 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.

Note: Queries using bitmap join index indicate the bitmap join index access path. The operation for bitmap join index is the same as bitmap index.

Viewing Partitioned Objects with EXPLAIN PLAN

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.

Examples of 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;
```

For the first example, consider the following statement:

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

Enter the following to display the EXPLAIN PLAN output:

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

Oracle displays something similar to the following:

```
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 the above example, 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.

For the next example, consider the following statement:

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

Finally, consider the following statement:

```
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 the above example, 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 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.

Examples of 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;
```

For the first example, consider the following statement:

```
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.
```

This example 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 the following example, the range partition row source iterates from partition 1 to 5, because no pruning is performed. 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.

```
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 the above example, 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.

Now consider the following statement:

```
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 the above example, 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.

Finally, consider the following statement:

```
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.
```

The last two examples are the same, except that `deptno` = 20 has been replaced by `deptno` = :dno. In this last 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.

Examples of Partial Partition-wise Joins

In the following 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;
    Table altered.
ALTER TABLE dept PARALLEL 2;
    Table altered.
```

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 PQ Distrib column contains the text PART (KEY), or partition key.

In the next example, 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;
    Table altered.
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.
```

Examples of 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.
```

Examples of 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:

```
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.

When the IN-List Column is an 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 RANGE  ALL                                KEY(INLIST)     KEY(INLIST)
INLIST ITERATOR
TABLE ACCESS     BY LOCAL INDEX 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.

When the IN-List Column is an Index and a 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 RANGE  ITERATOR                         KEY(INLIST)     KEY(INLIST)
TABLE ACCESS     BY LOCAL INDEX ROWID EMP         KEY(INLIST)     KEY(INLIST)
INDEX            RANGE SCAN           EMP_EMPNO   KEY(INLIST)     KEY(INLIST)
```

When the IN-List Column is a 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 RANGE   INLIST                       KEY(INLIST)       KEY(INLIST)
TABLE ACCESS      FULL           EMP           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
```

Example of 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 might 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 10, "Using SQL Trace and TKPROF"

PLAN_TABLE Columns

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

Table 9–1 PLAN_TABLE Columns **(Page 1 of 4)**

Column	Type	Description
STATEMENT_ID	VARCHAR2(30)	The value of the optional STATEMENT_ID parameter specified in the EXPLAIN PLAN statement.
TIMESTAMP	DATE	The date and time when the EXPLAIN PLAN statement was issued.
REMARKS	VARCHAR2(80)	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	VARCHAR2(30)	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 9–4 for more information on values for this column.

Table 9–1 PLAN_TABLE Columns **(Page 2 of 4)**

OPTIONS	VARCHAR2(225)	A variation on the operation described in the OPERATION column. See Table 9–4 for more information on values for this column.
OBJECT_NODE	VARCHAR2(128)	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.
OBJECT_OWNER	VARCHAR2(30)	The name of the user who owns the schema containing the table or index.
OBJECT_NAME	VARCHAR2(30)	The name of the table or index.
OBJECT_INSTANCE	NUMERIC	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	VARCHAR2(30)	A modifier that provides descriptive information about the object; for example, NON-UNIQUE for indexes.
OPTIMIZER	VARCHAR2(255)	The current mode of the optimizer.
SEARCH_COLUMNS	NUMBERIC	Not currently used.
ID	NUMERIC	A number assigned to each step in the execution plan.
PARENT_ID	NUMERIC	The ID of the next execution step that operates on the output of the ID step.
POSITION	NUMERIC	For the first row of output, this indicates the optimizer's estimated cost of executing the statement. For the other rows, it indicates the position relative to the other children of the same parent.
COST	NUMERIC	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. The value of this column is a function of the CPU_COST and IO_COST columns.
CARDINALITY	NUMERIC	The estimate by the cost-based approach of the number of rows accessed by the operation.
BYTES	NUMERIC	The estimate by the cost-based approach of the number of bytes accessed by the operation.
OTHER_TAG	VARCHAR2(255)	Describes the contents of the OTHER column. See Table 9–2 for more information on the possible values for this column.

Table 9–1 PLAN_TABLE Columns **(Page 3 of 4)**

PARTITION_START	VARCHAR2(255)	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 run time from partitioning key values. ROW LOCATION indicates that the start partition (same as the stop partition) will be computed at run 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	VARCHAR2(255)	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 run time from partitioning key values. ROW LOCATION indicates that the stop partition (same as the start partition) will be computed at run 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	NUMERIC	The step that has computed the pair of values of the PARTITION_START and PARTITION_STOP columns.
OTHER	LONG	Other information that is specific to the execution step that a user might find useful.
DISTRIBUTION	VARCHAR2(30)	Stores the method used to distribute rows from *producer* query servers to *consumer* query servers. See Table 9–3 for more information on the possible values for this column. For more information about consumer and producer query servers, see *Oracle9i Database Concepts*.

Table 9–1 PLAN_TABLE Columns **(Page 4 of 4)**

CPU_COST	NUMERIC	The CPU 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. The value of this column is proportional to the number of machine cycles required for the operation.
IO_COST	NUMERIC	The I/O 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. The value of this column is proportional to the number of data blocks read by the operation.
TEMP_SPACE	NUMERIC	The temporary space, in bytes, used by the operation as estimated by the optimizer's cost-based approach. For statements that use the rule-based approach, or for operations that don't use any temporary space, this column is null.

Table 9–2 describes the values that can appear in the OTHER_TAG column.

Table 9–2 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 9–3 describes the values that can appear in the DISTRIBUTION column:

Table 9–3 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 9–4 lists each combination of OPERATION and OPTION produced by the EXPLAIN PLAN statement and its meaning within an execution plan.

Table 9–4 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 "Viewing Bitmap Indexes with EXPLAIN PLAN" on page 9-10.
	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. The options column contain information supplied by a user-defined domain index cost function, if any.
FILTER		Operation accepting a set of rows, eliminates some of them, and returns the rest.
FIRST ROW		Retrieval of only the first row selected by a query.

Table 9–4 OPERATION and OPTION Values Produced by EXPLAIN PLAN **(Page 2 of 4)**

Operation	Option	Description
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.
	RANGE SCAN	Retrieval of one or more rowids from an index. Indexed values are scanned in ascending order.
(These are access methods.)	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.
NESTED LOOPS		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.
(These are join operations.)	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.

Table 9–4 OPERATION and OPTION Values Produced by EXPLAIN PLAN **(Page 3 of 4)**

Operation	Option	Description
	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 9–1 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 ACCESS	FULL	Retrieval of all rows from a table.
(These are access methods.)	SAMPLE	Retrieval of sampled rows from a table.
	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 RANGE	Retrieval of rows from a table based on a rowid range.
	SAMPLE BY ROWID RANGE	Retrieval of sampled rows from a table based on a rowid range.
	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.

Table 9–4 OPERATION and OPTION Values Produced by EXPLAIN PLAN **(Page 4 of 4)**

Operation	Option	Description
	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 might 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.

See Also: Chapter 1, "Introduction to the Optimizer"

10

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
- Sample TKPROF Output

Understanding SQL Trace and TKPROF

The SQL trace facility and TKPROF let you accurately assess the efficiency of the SQL statements an 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 10-4.

2. Enable the SQL trace facility for the desired session, and run the 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 10-5.

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 10-6.

4. Interpret the output file created in Step 3.

   See "Step 4: Interpreting TKPROF Output" on page 10-11.

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 10-16.

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:

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

Table 10–1 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 the 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 the 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 the operating system. The default value is the default destination for system dumps on the 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 can soon contain many hundreds of files, usually with generated names. It might 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 the operating system retains multiple versions of files, then be sure that the 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 can 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 the current *session*, enter the following:

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

> **Caution:** Because running the SQL trace facility increases system overhead, enable it only when tuning SQL statements, and disable it when you are finished.
>
> Setting SQL_TRACE to true can have a severe performance impact. For more information, see *Oracle9i Database Reference*.

Alternatively, you can enable the SQL trace facility for the 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 the session, enter:

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

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

> **Note:** You might need to modify the 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 the *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'
```

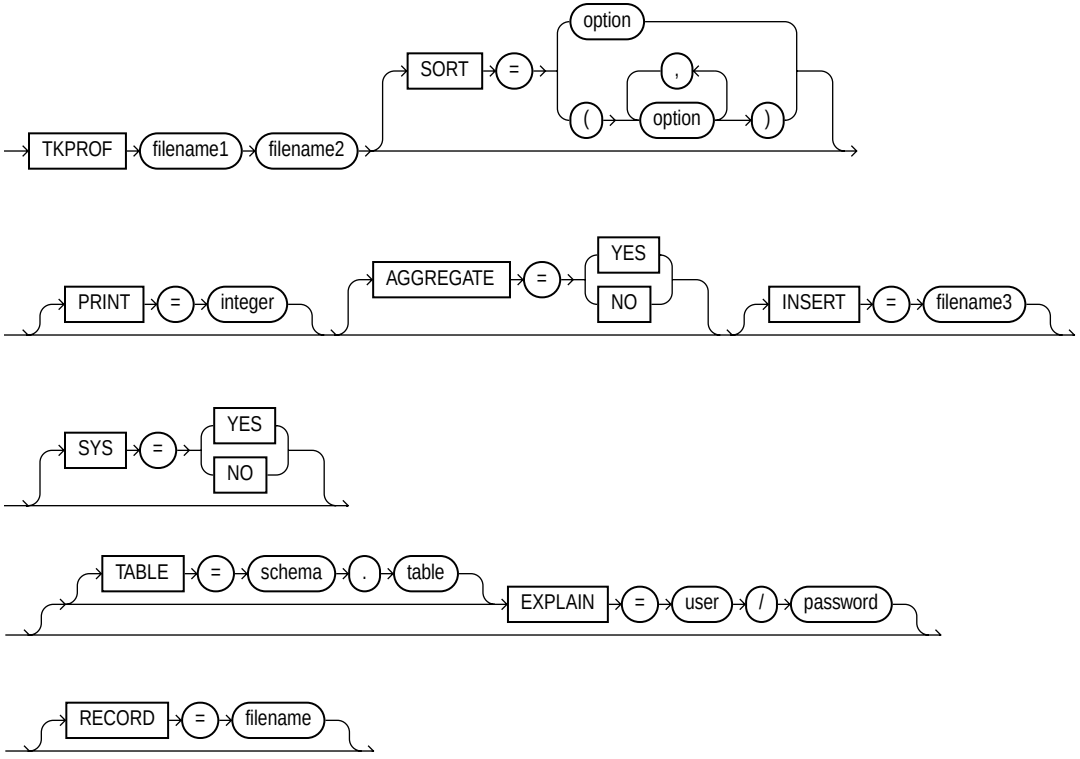
For this statement, TKPROF output includes the following information:

- 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

TKPROF::=



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

Use the following arguments with TKPROF:

Table 10–2 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 10–2 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 as follows:	
	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 10–2 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 *Oracle9i 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.

Examples of TKPROF Statement

This section provides two brief examples of TKPROF usage. For an complete example of TKPROF output, see "Sample TKPROF Output" on page 10-22.

TKPROF 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 10 statements in the trace file that have generated the most physical I/O:

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

TKPROF 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 the screen, and you might need to use continuation characters, depending on the 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 in TKPROF
- Library Cache Misses in TKPROF
- Statement Truncation in SQL Trace
- Identification of User Issuing the SQL Statement in TKPROF
- Execution Plan in TKPROF
- 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 in TKPROF

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. Usually, buffers are 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.

Interpreting the 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 might 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.

Understanding 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 (for example, 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, 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 "Avoiding the Trigger Trap" on page 10-22.

Library Cache Misses in TKPROF

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 10-6, the statement resulted in one library cache miss for the parse step and no misses for the execute step.

Statement Truncation in SQL Trace

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
```

Identification of User Issuing the SQL Statement in TKPROF

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 in TKPROF

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 9, "Using EXPLAIN PLAN" for 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: "Examples of TKPROF Statement" on page 10-10 for examples of finding resource intensive statements

With the exception of locking problems and inefficient PL/SQL loops, neither the CPU time nor the elapsed time is 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 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 might want to keep a history of the statistics generated by the SQL trace facility for an 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 might 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 10-6.

```
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:

- Avoiding the Argument Trap
- Avoiding the Read Consistency Trap
- Avoiding the Schema Trap
- Avoiding the Time Trap
- Avoiding the Trigger Trap

Avoiding 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, experiment with different data types in the query.

> **See Also:** "EXPLAIN PLAN Restrictions" on page 9-20 for information about TKPROF and bind variables

Avoiding 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)
```

Avoiding 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 might 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 following data:

```
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.

Avoiding the Time Trap

Sometimes, as in the following example, you might 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 might not need to be analyzed.

Avoiding 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.

If a DML statement appears to be consuming far more resources than you would expect, then check the tables involved in the statement for triggers and constraints that could be greatly increasing the resource usage.

Sample TKPROF Output

This section provides an extensive example of TKPROF output. Portions have been edited out for the sake of brevity.

Sample TKPROF 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
********************************************************************************
```

Sample TKPROF 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'
********************************************************************************
```

Sample TKPROF 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.
```

11

Using Autotrace in SQL*Plus

You can automatically get a report on the execution path used by the SQL optimizer and the statement execution statistics. The report is generated after successful SQL DML (that is, `SELECT`, `DELETE`, `UPDATE` and `INSERT`) statements. It is useful for monitoring and tuning the performance of these statements.

This chapter contains the following sections:

- Controlling the Autotrace Report
- Tracing Parallel and Distributed Queries

Controlling the Autotrace Report

You can control the report by setting the AUTOTRACE system variable.

`SET AUTOTRACE OFF`	No AUTOTRACE report is generated. This is the default.
`SET AUTOTRACE ON EXPLAIN`	The AUTOTRACE report shows only the optimizer execution path.
`SET AUTOTRACE ON STATISTICS`	The AUTOTRACE report shows only the SQL statement execution statistics.
`SET AUTOTRACE ON`	The AUTOTRACE report includes both the optimizer execution path and the SQL statement execution statistics.
`SET AUTOTRACE TRACEONLY`	Like SET AUTOTRACE ON, but suppresses the printing of the user's query output, if any.

To use this feature, you must have the PLUSTRACE role granted to you and a PLAN_TABLE table created in your schema.

See Also:

- *Oracle9i SQL Reference* for more information about the PLUSTRACE role and the PLAN_TABLE
- *SQL*Plus User's Guide and Reference* for more information about the AUTOTRACE variable of the SET command

Execution Plan

The execution plan shows the SQL optimizer's query execution path. Both tables are accessed by a full table scan, sorted, and then merged. Each line of the execution plan has a sequential line number. SQL*Plus also displays the line number of the parent operation.

The execution plan consists of four columns displayed in the following order:

Column Name	Description
ID_PLUS_EXP	Shows the line number of each execution step.
PARENT_ID_PLUS_EXP	Shows the relationship between each step and its parent. This column is useful for large reports.
PLAN_PLUS_EXP	Shows each step of the report.
OBJECT_NODE_PLUS_EXP	Shows the database links or parallel query servers used.

The format of the columns may be altered with the COLUMN command. For example, to stop the PARENT_ID_PLUS_EXP column being displayed, enter the following:

```
SQL> COLUMN PARENT_ID_PLUS_EXP NOPRINT
```

The default formats can be found in the site profile (for example, GLOGIN.SQL).

The execution plan output is generated using the EXPLAIN PLAN command.

> **See Also:** Chapter 9, "Using EXPLAIN PLAN" for more information about interpreting the output of EXPLAIN PLAN

Statistics

The statistics are recorded by the server when your statement executes and indicate the system resources required to execute your statement.

The *client* referred to in the statistics is SQL*Plus. *Oracle Net* refers to the generic process communication between SQL*Plus and the server, regardless of whether Oracle Net is installed.

You cannot change the default format of the statistics report.

> **See Also:** Chapter 3, "Gathering Optimizer Statistics" for more information about the statistics and how to interpret them

Example: Tracing Statements for Performance Statistics and Query Execution Path

If the SQL buffer contains the following statement:

```
SQL> SELECT D.DNAME, E.ENAME, E.SAL, E.JOB
  2  FROM EMP E, DEPT D
  3  WHERE E.DEPTNO = D.DEPTNO
```

The statement can be automatically traced when it is run:

```
SQL> SET AUTOTRACE ON
SQL> /

DNAME          ENAME             SAL JOB
-------------- ---------- ---------- ---------
ACCOUNTING     CLARK            2450 MANAGER
ACCOUNTING     KING             5000 PRESIDENT
ACCOUNTING     MILLER           1300 CLERK
RESEARCH       SMITH             800 CLERK
RESEARCH       ADAMS            1100 CLERK
RESEARCH       FORD             3000 ANALYST
RESEARCH       SCOTT            3000 ANALYST
RESEARCH       JONES            2975 MANAGER
SALES          ALLEN            1600 SALESMAN
SALES          BLAKE            2850 MANAGER
SALES          MARTIN           1250 SALESMAN
SALES          JAMES             950 CLERK
SALES          TURNER           1500 SALESMAN
SALES          WARD             1250 SALESMAN

14 rows selected.

Execution Plan
----------------------------------------------------------
0      SELECT STATEMENT Optimizer=CHOOSE
1    0   MERGE JOIN
2    1     SORT (JOIN)
3    2       TABLE ACCESS (FULL) OF 'DEPT'
4    1     SORT (JOIN)
5    4       TABLE ACCESS (FULL) OF 'EMP'
```

```
Statistics
----------------------------------------------------------
148  recursive calls
4  db block gets
24  consistent gets
6  physical reads
43  redo size
591  bytes sent via Oracle Net to client
256  bytes received via Oracle Net from client
3  Oracle Net roundtrips to/from client
2  sort (memory)
0  sort (disk)
14  rows processed
```

Note: Your output may vary depending on the version of the server to which you are connected and the configuration of the server.

Example: Tracing Statements Without Displaying Query Data

To trace the same statement without displaying the query data:

```
SQL> SET AUTOTRACE TRACEONLY
SQL> /

Execution Plan
----------------------------------------------------------
0      SELECT STATEMENT Optimizer=CHOOSE
1    0   MERGE JOIN
2    1     SORT (JOIN)
3    2       TABLE ACCESS (FULL) OF 'DEPT'
4    1     SORT (JOIN)
5    4       TABLE ACCESS (FULL) OF 'EMP'
```

```
Statistics
----------------------------------------------------------
0  recursive calls
4  db block gets
2  consistent gets
0  physical reads
0  redo size
599  bytes sent via Oracle Net to client
256  bytes received via Oracle Net from client
3  Oracle Net roundtrips to/from client
2  sort (memory)
0  sort (disk)
14  rows processed
```

This option is useful when you are tuning a large query, but do not want to see the query report.

Example: Tracing Statements Using a Database Link

To trace a statement using a database link:

```
SQL> SET AUTOTRACE TRACEONLY EXPLAIN
SQL> SELECT * FROM EMP@MY_LINK;

Execution Plan
----------------------------------------------------------
0      SELECT STATEMENT (REMOTE) Optimizer=CHOOSE
1    0   TABLE ACCESS (FULL) OF 'EMP'  MY_LINK.DB_DOMAIN
```

The execution plan shows the table being accessed on line 1 is via the database link `MY_LINK.DB_DOMAIN`.

Tracing Parallel and Distributed Queries

When you trace a statement in a parallel or distributed query, the execution plan shows the cost based optimizer estimates of the number of rows (the *cardinality*). In general, the cost, cardinality and bytes at each node represent cumulative results. For example, the cost of a join node accounts for not only the cost of completing the join operations, but also the entire costs of accessing the relations in that join.

Lines marked with an asterisk (*) denote a parallel or remote operation. Each operation is explained in the second part of the report.

The second section of this report consists of three columns displayed in the following order:

Column Name	Description
ID_PLUS_EXP	Shows the line number of each execution step.
OTHER_TAG_PLUS_EXP	Describes the function of the SQL statement in the OTHER_PLUS_EXP column.
OTHER_PLUS_EXP	Shows the text of the query for the Oracle Real Application Clusters database or remote database.

The format of the columns may be altered with the COLUMN command. The default formats can be found in the site profile (for example, GLOGIN.SQL).

Note: You must have Oracle7, Release 7.3 or greater to view the second section of this report.

Example: Tracing Statements With Parallel Query Option

To trace a parallel query running the parallel query option:

```
SQL> CREATE TABLE D2_T1 (UNIQUE1 NUMBER) PARALLEL -
>    (DEGREE 6);

Table created.

SQL> CREATE TABLE D2_T2 (UNIQUE1 NUMBER) PARALLEL -
>    (degree 6);

Table created.

SQL> CREATE UNIQUE INDEX D2_I_UNIQUE1 ON D2_T1(UNIQUE1);

Index created.

SQL> SET LONG 500 LONGCHUNKSIZE 500
SQL> SET AUTOTRACE ON EXPLAIN
SQL> SELECT /*+ INDEX(B,D2_I_UNIQUE1) USE_NL(B) ORDERED -
>    */ COUNT (A.UNIQUE1)
  2  FROM D2_T2 A, D2_T1 B
  3  WHERE A.UNIQUE1 = B.UNIQUE1;
```

SQL*Plus displays the following output:

```
Execution Plan
```

```
-------------------------------------------------------
0      SELECT STATEMENT Optimizer=CHOOSE (Cost=1
                        Card=263 Bytes=5786)
1    0   SORT (AGGREGATE)
2    1     NESTED LOOPS* (Cost=1 Card=263 Bytes=5785)
                                                :Q8200
3    2       TABLE ACCESS* (FULL) OF 'D2_T2'    :Q8200
4    2       INDEX* (UNIQUE SCAN) OF 'D2_I_UNIQUE1'
                    (UNIQUE)                    :Q8200
2 PARALLEL_TO_SERIAL  SELECT /*+ ORDERED NO_EXPAND
                      USE_NL(A2) INDEX(A2) PIV_SSF */
                      COUNT(A1.C0) FROM (SELECT/*+
                      ROWID(A3) */ A3."UNIQUE1" FROM
                      "D2_T2" A3 WHERE ROWID BETWEEN :1
                      AND :2) A1, "D2_T1" A2 WHERE
                      A1.C0=A2."UNIQUE1"
3 PARALLEL_COMBINED_WITH_PARENT
4 PARALLEL_COMBINED_WITH_PARENT
```

Line 0 of the execution plan shows the cost based optimizer estimates the number of rows at 263, taking 5786 bytes. The total cost of the statement is 1.

Lines 2, 3 and 4 are marked with asterisks, denoting parallel operations. For example, the nested loops step on line 2 is a PARALLEL_TO_SERIAL operation. PARALLEL_TO_SERIAL operations execute a SQL statement to produce output serially. Line 2 also shows that the parallel query server had the identifier *Q8200*.

Monitoring Disk Reads and Buffer Gets

Monitor disk reads and buffer gets by executing the following statement in SQL*Plus:

```
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' is high relative to the amount of data returned, then this is 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.

Note: You can also monitor disk reads and buffer gets using V$SQL or TKPROF.

12

Using Oracle Trace

This chapter describes how to use Oracle Trace to collect Oracle server event data.

The topics in this chapter include:

- Overview of Oracle Trace
- Collecting Oracle Trace Data
- Accessing Oracle Trace Collection Results
- Oracle Server Events
- Troubleshooting Oracle Trace

Overview of Oracle Trace

Oracle Trace is a general-purpose event-driven data collection product, which the Oracle server uses to collect performance and resource utilization data, such as SQL parse, execute, and fetch statistics, and wait statistics.

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 instrumented with the Oracle Trace API. That is, the product has embedded calls to the Oracle Trace API. 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 product. An example of a point event is an error occurrence.

- Duration events

  Duration events have a beginning and an 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 has more than a dozen events. The following are three of these events:

- 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.

Event Sets

Oracle Trace events can be organized into **event sets** that restrict the data collection to specific events. You can establish event sets for performance monitoring, auditing, diagnostics, or any logical event grouping.

Each event set is described by its own **product definition file** (`.fdf`). The product definition file is a set of events and their associated data items. The complete set of

events defined for an instrumented product is referred to as the ALL event set. Other event sets are then derived from the ALL set. For example, the Oracle Server includes an event set known as the EXPERT set. This set includes SQL event data used by the Oracle Expert tuning application, but excludes other events, such as wait events.

Accessing Collected Data

During a collection, Oracle Trace stores event data in memory and periodically writes it to a collection binary file. This method ensures low resource overhead associated with the collection process. You can access event data collected in the binary file by formatting the data to database tables, which makes the data available for fast, flexible access. These database tables are called Oracle Trace formatter tables.

Collecting Oracle Trace Data

You can collect Oracle Trace data using one of the following mechanisms:

- Oracle Trace Command-Line Interface (CLI)
- Database `ORACLE_TRACE_*` initialization parameters
- Oracle Trace stored procedures run from PL/SQL

Using the Oracle Trace Command-Line Interface

You can control Oracle Trace server collections with the Oracle Trace CLI (command-line interface). The CLI is invoked by the `OTRCCOL` executable for the following functions:

- `OTRCCOL START` *`job_id input_parameter_file`*
- `OTRCCOL STOP` *`job_id input_parameter_file`*
- `OTRCCOL CHECK` *`collection_name`*
- `OTRCCOL FORMAT` *`input_parameter_file`*
- `OTRCCOL DCF` *`col_name`* `cdf_file`
- `OTRCCOL DFD` *`col_name`* username password service [*`col_id`*]

The `job_id` parameter should be set to a value of 1.

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.

> **Note:** The server parameter ORACLE_TRACE_ENABLE must be set to true to allow Oracle Trace to collect any server event data. This is required for Oracle8 and newer servers.

Starting Collections

The OTRCCOL START function invokes a collection based on parameter values contained in the input parameter file. For example:

```
OTRCCOL START 1 my_start_input_file
```

where file my_start_input_file contains at least the following input parameters:

```
col_name= <collection name>
cdf_file= <collection name>.cdf
dat_file= <collection name>.dat
fdf_file= <facility definition file>.fdf
```

The server event sets that can be used as values for the fdf_file parameter are ORACLE, ORACLEC, ORACLED, ORACLEE, and ORACLESM, plus CONNECT, SQL_ONLY, SQL_PLAN, SQL_TXN, SQLSTATS, SQLWAITS, and WAITS.

> **See Also:** Table 12–2 for a description of the server event sets

Collection .cdf and .dat files are created in the directory $ORACLE_HOME/otrace/admin/cdf by default for collections started using the CLI (or PL/SQL procedures), unless overridden by EPC_COLLECTION_PATH environment variable.

> **Note:** This chapter refers to 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 *Oracle9i Database Reference*.

The <*collections name*> parameter can be any valid unique filename.

For Oracle database collections, one additional parameter is required:

```
regid= 1 192216243 0 0 5 <database SID>
```

> **Note:** Older CLI versions required this syntax exactly, with no whitespace before '=' but at least some whitespace after '='. This is no longer true: CLI is not whitespace sensitive.

The `regid` parameter record identifies a database by SID where Oracle Trace collection is to be performed. The six elements making up the `regid` parameter record are as follows, in this order:

- flag (this should be 1 for this usage)
- vendor number (Oracle's vendor number is 192216243)
- `cf_num` (more on this below)
- `cf_val` (more on this below)
- facility number (the Oracle server is facility number 5)
- database SID

The `cf_num` and `cf_val` elements should set to zero in this basic Oracle database collection `regid` above.

Limiting Data Collected

There are several ways of limiting the amount of data collected. For example, additional `regid` records can be specified to reduce the amount of collected data, and nonzero `cf_num` and `cf_val` can be specified in those situations. In the Oracle server, Oracle Trace cross facility item 6 (`cf_num` = 6) is reserved to record database userID values.

Restrict By User

In the Oracle server, Oracle Trace cross facility item 6 (`cf_num` = 6) is reserved to record database userID values. So, for example, if you provide an additional `regid` record with `cf_num` = 6 and `cf_val` = <*some DB userID*>, then the collection of database event data is limited to only those events performed by that database user.

If you are interested only in collecting database activity for users 23 and 45, then you would provide the following 3 `regid` records:

```
regid= 1 192216243 0  0 5 ORCL
regid= 1 192216243 6 23 5 ORCL
regid= 1 192216243 6 45 5 ORCL
```

Restrict By Process

The input parameter file used by the CLI when starting a collection can also contain the following optional parameters, for both database and nondatabase Oracle Trace collections:

```
prores= <process restriction>
max_cdf= <maximum collection file size>
```

If no process restriction records are specified, then there are no restrictions on which processes can take part in the collection. If process restrictions are used, then one or more process ID (PID) values can be specified, as well as the operating system username for the owner of each process of interest.

Setting a Limit on File Size

The `max_cdf` parameter is often useful, in several different modes of use. This parameter specifies the maximum amount of Oracle Trace data that should be collected, in bytes (in other words, size of the <*collection*>.dat file).

A zero value indicates that no limit should be imposed; otherwise, a positive value up to 2 GB can be specified to stop the data collection when that size limit is reached. In addition, a negative value can be specified (but not less than -2 GB), which instructs Oracle Trace to collect data in its "circular data file" mode: when <*collection*>.dat reaches magnitude(`max_cdf`), then save that data (and delete any previously saved dat file), and then start collecting to a new <*collection*>.dat file. This limits the total amount of disk space used, but allows Oracle Trace data collection to continue until you manually stop collection.

Checking the Status of a Collection

Verify that the collection was started.

```
otrccol check <collection_name>
```

The collection should show as active, not active, or not found.

Stopping Collections

The `OTRCCOL STOP` function halts a running collection as follows:

```
OTRCCOL STOP 1 my_stop_input_file
```

where `my_stop_input_file` contains the collection name and `cdf_file` name.

Formatting Collections

The `OTRCCOL FORMAT` function 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 `full_format` value of 0 produces a partial format, which only formats new data; in other words, data collected since any previous format.

> **See Also:** "Formatting Oracle Trace Data to Oracle Tables" on page 12-13 for more information on formatting an Oracle Trace collection using the `otrcfmt` utility program.

Deleting Collections

The `OTRCCOL DCF` (delete collection files) function deletes collection `.cdf` and `.dat` files for a specific collection. The `OTRCCOL DFD` (delete formatted data) function deletes formatted data from the Oracle Trace formatter tables for a specific collection. You can specify an optional `col_id` parameter for a selective DFD, where more than one `col_id` has been created for a collection by multiple (full) formats.

Using Initialization Parameters to Control Oracle Trace

Six Oracle database initialization parameters are set up by default to control Oracle Trace. By logging into a privileged account in the database and executing the `SHOW PARAMETER ORACLE_TRACE` statement, you see the following parameters:

Table 12–1 Oracle Trace Initialization Parameters

Name	Type	Default 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 these Oracle Trace server parameters to allow Oracle Trace collection of server event data and use them by adding them to the initialization file.

However, this method for controlling the Oracle Trace collection is rather inflexible: the collection name cannot be changed without performing a database shutdown. (For Oracle releases prior to 8.1.7, the collection can only be stopped by doing a shutdown, then setting ORACLE_TRACE_ENABLE = false before restarting.) However, with ORACLE_TRACE_ENABLE = true but ORACLE_TRACE_COLLECTION_NAME = "" [that is, empty name string], Oracle Trace collections of database event data can be performed using one of the other collection control mechanisms; for example, the Oracle Trace CLI. These other mechanisms are more flexible than the database initialization parameters. In general, they are preferred over using parameters for collection control.

Enabling Oracle Trace Collections

The ORACLE_TRACE_ENABLE database initialization parameter is false by default. This disables any collection of Oracle Trace data for that server, regardless of the mechanism used to control the collection.

Setting ORACLE_TRACE_ENABLE to true in <*DBinit*>.ora enables Oracle Trace collections for the server, but it does not necessarily start a collection when the database instance is started. If the database parameters alone are to be used to start an Oracle Trace collection of database event data, then all 6 ORACLE_TRACE_* parameters must be specified, or have nonnull values by default. Typically, this means that both ORACLE_TRACE_ENABLE must be set to true and a nonnull ORACLE_TRACE_COLLECTION_NAME must be provided (up to 16 characters in length).

> **Note:** The collection name is also used to form the <*collection name*>.cdf and .dat binary file names, so 8.3 file naming conventions may apply on some platforms. 8.3 file naming means systems where filenames are restricted to 8 or fewer characters, plus a file extension of 3 or fewer characters.

ORACLE_TRACE_ENABLE is now a dynamic parameter (as of Oracle8*i*, Release 3), so it can be set to true or false while the database is running. This can be done for the current database session or for all sessions (including future ones), using ALTER SESSION or ALTER SYSTEM statements. When the database is subsequently shut down and then restarted, the <*DBinit*>.ora setting for ORACLE_TRACE_ENABLE is again used to initially enable or disable Oracle Trace collection of database event data.

Determining the Event Set that Oracle Trace Collects

The ORACLE_TRACE_FACILITY_NAME database initialization parameter specifies the event set that Oracle Trace collects, if the database parameters are used to control data collection. The default for this parameter is ORACLED (in other words, Oracle "default" event set).

> **Note:** The ORACLE_TRACE_FACILITY_NAME parameter does not use a file extension. So, the .fdf extension should not be specified as part of this parameter.

With database parameters set to start an Oracle Trace Collection, if the database does not begin collecting data, then check the following:

- The event set file, identified by ORACLE_TRACE_FACILITY_NAME, with an .fdf extension, 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 the Oracle Trace admin directory: COLLECT.DAT, FACILITY.DAT (or PROCESS.DAT for Oracle 7.3), and REGID.DAT. If they do not, then run the OTRCCREF executable to create or recreate 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 if the Oracle Trace Collection Services OTRCCOL image must log an error.
- Look for *.trc files in the directory specified by the server USER_DUMP_DEST initialization parameter. Searching for "epc" in the *.trc files might give errors. These errors and their descriptions may be found in $ORACLE_HOME/otrace/mesg/epcus.msg (assuming US installation), depending on availability for a given platform.

Controlling Oracle Trace Collections from PL/SQL

Oracle provides an additional Oracle Trace library that allows control of both database and nondatabase Oracle Trace collections from PL/SQL.

Note: Older versions of the Oracle server, back to release 7.3.3, provided other stored procedures to start and stop Oracle Trace database collections, but with significant limitations. For example, only database sessions already active when the collection was started participated in the collection: no database event data was collected for sessions that began after the Oracle Trace collection started.

As of Oracle8, these limitations were eliminated for database collections started via the Oracle Trace CLI. However, these limitations still applied to database collections controlled via the older Oracle7 stored procedures. The new procedures provided with Oracle Trace 8.1.7 remove these limitations, permitting the same level of collection control as the Oracle Trace CLI, and for both database and nondatabase collections.

Both the name and the location of this new library are platform-dependent. On Unix platforms, the library is $ORACLE_HOME/lib/libtracepls9.so.

On Win32 platforms (for example, Windows NT), the library is %ORACLE_HOME%\bin\oratracepls9.dll.

The otrace/admin directory contains two new SQL scripts that can be used to define a database LIBRARY object for this library, and to define the procedures that can be used to call out to the library from PL/SQL:

- OTRCPLSLIB.SQL
- OTRCPLSCMD.SQL

> **Note:** By default, OTRCPLSLIB.SQL grants access to the LIBRARY to all database users. You can edit this SQL script to restrict access as appropriate.

In addition, the otrace/demo directory contains several SQL scripts showing PL/SQL examples that start, stop, and then format an Oracle Trace collection. These are respectively:

- OTRCPLSSC.SQL
- OTRCPLSCC.SQL
- OTRCPLSFC.SQL

In the "start collection" example script OTRCPLSSC.SQL, the regid_list contains only a single element: "1 192216243 0 0 5 ORCL". The inner double quotes are required to form a single regid string from its six components. These components are the following, in the order shown:

- flag (this should be 1 for this usage)
- vendor number (Oracle's vendor number is 192216243)
- cf_num (see below)
- cf_val (see below)
- facility number (Oracle server is facility number 5)
- database SID

For an Oracle Trace database collection, a regid string like this example is required, basically to identify the database SID and to specify that you are collecting for an Oracle server. The cf_num and cf_val should be zero in this basic regid record.

Additional regid records can be specified in order to reduce the amount of collected data. This is when the cf_num and cf_val items are used. In the Oracle server, Oracle Trace cross facility item 6 (cf_num = 6) is reserved to record database

userID values. So, if you provide an additional `regid` record with `cf_num` = 6 and a `cf_val` = <*some DB userID*>, then the collection of database event data is limited to only those events performed by that database user. For example, if you are only interested in collecting database activity for users 23 and 45, then the `regid_list` consists of three records:

```
regid_list   VARCHAR2(256) := '"1 192216243 0  0 5 ORCL",
                               "1 192216243 6 23 5 ORCL",
                               "1 192216243 6 45 5 ORCL"';
```

Note: For better readability, the layout of this `regid` string has been simplified.

Similarly, the `fdf_list` argument could specify the name of a single `.fdf` file (facility definition file). Typically, this is the case. However, more than one `.fdf` could be specified in `fdf_list` if multiple facilities are involved in the collection. Of course, only one `.fdf` can be specified for any given facility; for example, the database.

On the other hand, the process restriction list `prores_list` can be empty. This indicates that there are to be no restrictions on which processes can take part in the collection. If process restrictions are used, then one or more process ID (PID) values can be specified, as well as the operating system username for the owner of each process.

Other arguments in the "start collection" example in `OTRCPLSSC.SQL` are single numeric or string values, as shown. For example, the collection name and maximum collection data file size specified by the `col_name` and `maxsize` variables, respectively.

Accessing Oracle Trace Collection Results

Running an Oracle Trace collection produces the following collection files:

- <*collection name*>`.cdf` is the binary Oracle Trace collection definition file for the collection. It describes what is to be collected.
- <*collection name*>`.dat` is the output file containing the collected Oracle Trace event data in binary form. When newly created, the empty `.dat` file contains only 35 bytes of file header information.

You can access the Oracle Trace data in the collection files in the following ways:

- The data can be formatted to Oracle tables for SQL access and reporting.
- You can create Oracle Trace reports from the binary file.

Formatting Oracle Trace Data to Oracle Tables

You can format Oracle Trace binary collection data to Oracle database tables, and you can then access this formatted data using SQL or other tools. The Oracle Trace format produces a separate table for each event type collected; for example, a parse event table is created to store data for all database parse events that were recorded during a server collection.

> **Note:** For Oracle server releases 7.3.4 and later, the Oracle Trace formatter automatically creates the formatter tables as needed.

Use the following syntax to format an Oracle Trace collection with the `OTRCFMT` formatter utility:

```
OTRCFMT [options] <collection_name>.cdf [user/password@database]
```

If collection `.cdf` and `.dat` are not located in the current default directory, then specify the full file path for the `.cdf` file.

If you omit `user/password@database` (or any part of it, such as password or database), then `OTRCFMT` 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` (which generate a new collection ID in the formatter tables).

Oracle Trace provides several sample SQL scripts that you can use to access the formatted server event data tables. These are located in `OTRCRPT*.SQL` in the otrace directory tree.

> **Note:** Because there are incompatibilities between at least some versions of the formatter tables, use a separate database schema for each version of the Oracle Trace formatter.

Running the Oracle Trace Reporting Utility

The Oracle Trace reporting utility displays data for items associated with each occurrence of a server event. These reports can be quite large. You can control the report output by using optional statement qualifiers. Use the following report utility statement syntax:

```
OTRCREP [options] <collection name>.cdf
```

If collection .cdf and .dat are not located in the current default directory, then specify the full file path for the .cdf file.

First, you might 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 qualifiers:

output_path	Specifies a full output path for the report files. If this path is not specified, then the files are placed in the current directory.
-p[<*pid*>]	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 PID, then you have one file for each process that participated in the collection. The output files are named <*collection_name*>_Ppid.txt.
-P	Produces a report file name PROCESS.txt that lists all processes that participated in the collection. It does not include event data. You can produce this report first to determine the specific processes for which you 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.
-a	Creates a report containing all the events for all products, in the order they occur in the data collection (.dat) file.

Default OTRCREP report output, with no optional qualifiers specified, consists of one text file per event type collected. Data from all participating processes are combined in each of these text files.

Oracle Server Events

The following sections describe events that have been instrumented in Oracle Server. Most of the events are useful for performance analysis and tuning and workload analysis by Oracle Expert.

There are two types of events: **point** events and **duration** 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 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, the occurrence of an error within a transaction.

Table 12–2 lists the Oracle Server events instrumented for Oracle Trace. For more detailed descriptions, refer to the section for the event in which you are interested.

Table 12–2 Oracle Server Events

Event	Description	Type of Event
1 Connection	Connection to a database.	Point
2 Disconnect	Disconnection from a database.	Point
3 ErrorStack	Code stack for core dump.	Point
4 Migration	Session migration between shared server processes.	Point
5 ApplReg	Application context information.	Point
6 RowSource	Row information. For Oracle Server release 8.0.2 and higher, this also includes data about the execution plan.	Point
7 SQLSegment	Text of SQL statement.	Point
8 Parse	SQL parsing information.	Duration
9 Execute	Information for execution of SQL.	Duration
10 Fetch	Actual row retrieval information.	Duration
11 LogicalTX	The first time a database command is performed that may change the database status.	Duration
12 PhysicalTX	Event marking a definite change in database status.	Duration
13 Wait	Generic WAIT event. Context is provided in the event strings.	Point

Data Items Collected for Events

Specific kinds of information, known as items, are associated with each event. There are three types of items:

- Resource Utilization Items
- Cross-Product Items
- Items Specific to Oracle Server Events

Resource Utilization Items

Oracle Trace has a standard set of items, called resource utilization items, that it can collect for any instrumented application, including the Oracle Server. In addition, all duration events in the Oracle Server include items for database statistics specific to the Oracle Server.

The standard resource utilization items are described in Table 12–3.

An Oracle Trace collection can be formatted into Oracle tables for access, analysis, and reporting. The last column contains the data type for data items formatted to the Oracle database.

Table 12–3 Standard Resource Utilization Items

Item Name	Description	Item ID	Datatype of Formatted Data
UCPU	Amount of CPU time in user mode	129	number
SCPU	Amount of CPU time in system mode	130	number
INPUT_IO	Number of times file system performed input	131	number
OUTPUT_IO	Number of times file system performed output	132	number
PAGEFAULTS	Number of hard and soft page faults	133	number
PAGEFAULT_IO	Number of hard page faults	134	number
MAXRS_SIZE	Maximum resident set size used	135	number

The implementation of the item is platform specific; if the item is not implemented, the value is 0. For example, currently only CPU times are recorded on Windows NT.

Cross-Product Items

Oracle Trace provides a set of 14 items called cross-product items (also known as cross-facility items for historical reasons). These data items allow programmers to relate events for different products. For example, a business transaction may generate events in two products: an application and the database. The cross-product data items allow these disparate events to be joined for analysis of the entire business transaction.

Cross-product items are reserved for specific products or product types as described in Table 12–4. If you do not use the products for which items are reserved, then you can use those items for your own purposes.

Table 12–4 Cross-Product Items

Item Name	Layer	Description	Item ID	Datatype of Formatted Data
CROSS_FAC_1	Application	Application ID. For use by high-level applications such as Oracle Financials, third-party or customer applications	136	number
CROSS_FAC_2	Oracle Forms	Oracle Forms ID	137	number
CROSS_FAC_3	Oracle Net	Remote node connection ID	138	number
CROSS_FAC_4	Oracle Server	Transaction ID	139	number
CROSS_FAC_5	Oracle Server	Hash_ID of SQL statement	140	number
CROSS_FAC_6	Oracle Server release 8.x	User ID	141	number
CROSS_FAC_7	Oracle Server release 8.x	Wait type	142	number
CROSS_FAC_8	n/a	Not reserved	143	number
CROSS_FAC_9	n/a	Not reserved	144	number
CROSS_FAC_10	n/a	Not reserved	145	number
CROSS_FAC_11	n/a	Not reserved	146	number
CROSS_FAC_12	n/a	Not reserved	147	number
CROSS_FAC_13	n/a	Not reserved	148	number
CROSS_FAC_14	n/a	Not reserved	149	number

Note: In this version of Oracle Trace, the term "facility" has been changed to "product". Therefore, the items named CROSS_FAC_*x* are cross-product items.

Cross-product item 1 (referred to as CROSS_FAC_1) contains data only if data is supplied by an instrumented application.

Cross-product item 2 (CROSS_FAC_2) is reserved for use by a future release of Oracle Forms. Instrumented applications and Oracle Forms pass identification data to the Oracle Server collection through these cross-product items.

Cross-product item 3 (CROSS_FAC_3) is reserved for use by Oracle Net. Oracle Net supplies the connection ID to Oracle Trace through CROSS_FAC_3. CROSS_FAC_3 is the key element in coordinating client/server or multitier Oracle Trace collections. Oracle Trace uses the Oracle Net global connection ID as the common element to match in the merger, for example the client and server collection files. The global connection ID is the same for the client and the server connection.

Most Oracle Server events record cross-product items 1 through 6. (Cache I/O does not.)

Items Specific to Oracle Server Events

The Oracle Server product (or facility) definition files (that is, *.fdf) defines items specific to the Oracle Server. Use the item's number to locate it within the list. The formatted datatype describes how the Oracle Trace formatter defines the item when it formats data into an Oracle database.

The Oracle Server items are listed in Table 12–5.

Table 12–5 Oracle Server Items

Item Name	Description	Item Number	Formatted Datatype
App_Action	Action name set by using the DBMS_APPLICATION_INFO.SET_MODULE procedure	23	varchar2 (1020)
App_Module	Module name set using the DBMS_APPLICATION_INFO.SET_MODULE procedure	22	varchar2 (1020)
Commit_Abort	Indicates if a transaction committed or aborted	24	number

Table 12–5 Oracle Server Items (Cont.)

Item Name	Description	Item Number	Formatted Datatype
Consistent_ Gets	Number of blocks retrieved in consistent mode (did not change the data and therefore did not create any locks or conflicts with other users)	104	number
CPU_Session	CPU session	112	number
Current_UID	Current user ID	36	number
Cursor_Number	Number of cursor associated with SQL statement	25	number
DB_Block_ Change	Number of blocks changed	102	number
DB_Block_Gets	Number of blocks retrieved in current mode. For large queries, this item tells how many sections of the database (logical pages) were fetched to retrieve all needed records.	103	number
Deferred_ Logging	Value used by Oracle Trace internally	14	number
Depth	Recursive level at which SQL statement is processed	32	number
Description	Depends upon event in which it occurs (for example, wait event description)	43	varchar2 (1020)
Elapsed_ Session	Elapsed time for the session	113	number
End_of_Fetch	Flag set if data retrieved is last data from query	38	number
Lib_Cache_Addr	Address of SQL statement in library cache	27	varchar2(64)
Login_UID	Internal ID within the Oracle database that identifies the user ID for the session	15	number
Login_UName	Internal ID within the Oracle database that identifies the system account name for the session	16	varchar2 (1020)
Missed	Flag set if SQL statement was missing in library cache	33	number
Object_ID[1]	Object ID of the row source	46	number

Table 12–5 Oracle Server Items (Cont.)

Item Name	Description	Item Number	Formatted Datatype
Operation[1]	Text of the operation	47	varchar2 (1020)
Operation_ID[1]	Position of the operation within the execution plan for a statement	28	number
Optimizer_Mode	Oracle optimizer mode	35	varchar2(128)
Oracle_Cmd_Type	Oracle command number	34	number
Oracle PID	Oracle process ID	11	number
OS_Image	Operating system image (program name)	42	long
OS_Mach	Operating system host machine	20	varchar2 (1020)
OS_Term	Operating system terminal	19	varchar2 (1020)
OS_UName	Operating system username	18	varchar2 (1020)
P1	The definition of P1 depends upon the event in which it occurs.	1	number
P2	The definition of P2 depends upon the event in which it occurs.	2	number
P3	The definition of P3 depends upon the event in which it occurs.	3	number
P4	The definition of P4 depends upon the event in which it occurs.	4	number
P5	The definition of P5 depends upon the event in which it occurs.	5	number
P6	The definition of P6 depends upon the event in which it occurs.	6	number
P7	The definition of P7 depends upon the event in which it occurs.	7	number

Table 12–5 Oracle Server Items (Cont.)

Item Name	Description	Item Number	Formatted Datatype
P8	The definition of P8 depends upon the event in which it occurs.	8	number
P9	The definition of P9 depends upon the event in which it occurs.	9	number
P10	The definition of P10 depends upon the event in which it occurs.	10	number
Parent_Op_ID[1]	Parent operation	44	number
PGA_Memory	Process Global Area memory	101	number
Physical Reads	Number of blocks read from disk	105	number
Position[1]	Position within events having same parent operation	45	number
Position_ID[2]	Position of the operation within the execution plan for a statement	28	number
Redo_Entries	Number of redo entries made by process	106	number
Redo_Size	Size of redo entries	107	number
Row_Count	Number of rows processed	29	number
Schema_UID	Schema user ID	37	number
Session_Index	Oracle session ID	12	number
Session_Serial	Session serial number	13	number
SID	Text version of session ID	17	varchar2 (1020)
Sort_Disk	Number of disk sorts performed	110	number
Sort_Memory	Number of memory sorts performed	109	number
Sort_Rows	Total number of rows sorted	111	number
SQL_Text	Text of SQL statement	31	long
SQL_Text_Hash	Pointer to SQL statement	26	number
SQL_Text_ Segment	Address of SQL text	30	number

Table 12–5 Oracle Server Items (Cont.)

Item Name	Description	Item Number	Formatted Datatype
T_Scan_Rows_Got	Rows processed during full table scans	108	number
TX_ID	Unique identifier for a transaction that consists of rollback segment number, slot number, and wrap number	41	varchar2(72)
TX_SO_Addr	The address of the transaction state object	40	varchar2(64)
TX_Type	Type of the transaction. Value is a bitmap (for example, 2 active transaction, 0X10 space transaction, 0X20 recursive transaction).	39	number
UGA_Memory	User Global Area session memory	100	number
Wait_Time	Elapsed time, in hundredths of seconds, for the wait event	21	number

[1] Item specific to Oracle Server release 8.0.2 and higher

[2] Replaced by Operation_ID for Oracle Server release 8.0.2 and higher

Items Associated with Each Event

The following sections describe each event in more detail and provide tables that list the items associated with each event. For item descriptions, refer to Table 12–5.

Note: Prior to Oracle Server release 8.0.5, cross-product items 1-5 were set by the server code. Starting with Oracle Server release 8.0.5, cross-product item 6 was added (and cross-product item 7 for wait events.)

When you format data, Oracle Trace creates a table for each event type collected. The name of the event data table is **V_vendor#_F_product#_E_event#_version**, where **version** is the number of the Oracle Server release. Any periods in the product version are replaced with underscores. You can use the otrcsyn.sql script to create synonyms for these tables.

Note: The following tables use Oracle7 Server names for example purposes.

The Oracle Trace formatter creates a column for each event item. For point events, the column name is the same as the item name. For duration events, the items for the start event have _START appended to the item name and the items for the end event have _END appended to the item name.

The formatter automatically includes additional columns for collection number, process identifier, and timestamp information as described in Table 12–6.

Table 12–6 Additional Columns Included by Oracle Trace Formatter

Column Name	Description	Datatype
collection_number	collection ID, automatically assigned by the formatter	number(4)
epid	process ID	number(8)
timestamp	logged time for point events	date
timestamp_nano	fraction of seconds of logged time for point events	number
timestamp_start	duration event start time	date
timestamp_nano_start	fraction of seconds of duration event start time	number
timestamp_end	duration event end time	date
timestamp_nano_end	fraction of seconds of duration event end time	number

Event Statistics Block

Items relating to database performance appear in several events. For convenience, these items are referenced as the Event Statistics Block. The items in the Event Statistics block are shown in Table 12–7:

Table 12–7 Event Statistics Block

UGA_Memory	PGA_Memory	DB_Block_Change
DB_Block_Gets	Consistent_Gets	Physical_Reads
Redo_Entries	Redo_Size	T_Scan_Rows_Got
Sort_Memory	Sort_Disk	Sort_Rows
CPU_Session	Elapsed_Session	

Connection Event

The Connection event (event=1) records every time a connection is made to a database. Items associated with the Connection event are shown in Table 12–8. The

name of the formatter table is V_192216243_F_5_E_1_7_3 (for Oracle Server release 7.3).

Table 12–8 Items Associated with the Connection Event

Session_Index	Session_Serial	Oracle_PID
Login_UID	Login_UName	SID
OS_UName	OS_Term	OS_Mach
OS_Image	Cross-Product Items 1-6	

The Oracle Server uses the combination of Session_Index and Session_Serial to uniquely identify a connection. Oracle Net uses the connection ID, stored in CROSS_FAC_3, to uniquely identify a connection.

Disconnect Event

The Disconnect event records every time a database disconnection is made. Items associated with the Disconnect event are shown in Table 12–9. The name of the formatter table is V_192216243_F_5_E_2_7_3.

Table 12–9 Items Associated with the Disconnect Event

Session_Index	Session_Serial	Event Statistics Block
Oracle_PID	Cross-Product Items 1-6	

A Disconnect event corresponds to at most one Connection event. Therefore, the same fields uniquely identify a disconnect: either the combination of Session_Index and Session_Serial, or CROSS_FAC_3.

ErrorStack Event

The ErrorStack event identifies the process that has the error. Items associated with the ErrorStack event are shown in Table 12–10. The name of the formatter table is V_192216243_F_5_E_3_7_3.

Table 12–10 Items Associated with the ErrorStack Event

Session_Index	Session_Serial	Oracle_PID
P1	P2	P3
P4	P5	P6
P7	P8	Cross-Product Items 1-6

The ErrorStack event does not have an explicit identifier. The combination of Session_Index, Session_Serial, Timestamp, and Timestamp_Nano should uniquely identify a specific ErrorStack event.

Migration Event

The Migration event is logged each time a session migrates to a shared server process. The name of the formatter table is V_192216243_F_5_E_4_7_3. This event is currently disabled in the Oracle server code.

Items associated with the Migration event are shown in Table 12–11.

Table 12–11 Items Associated with the Migration Event

Session_Index	Session_Serial	Oracle_PID
Cross-Product Items 1-6		

The Migration event does not have an explicit identifier. The combination of Session_Index, Session_Serial, Timestamp, and Timestamp_Nano should uniquely identify a specific Migration event.

ApplReg Event

The ApplReg event (event=5) registers with Oracle Trace where the application is at a certain point in time. Items associated with the ApplReg event are shown in Table 12–12. The name of the formatter table is V_192216243_F_5_E_5_7_3.

Table 12–12 Items Associated with the ApplReg Event

Session_Index	Session_Serial	App_Module
App_Action	Cross-Product Items 1-6	

The ApplReg event does not have an explicit identifier. The combination of Session_Index, Session_Serial, Timestamp, and Timestamp_Nano should uniquely identify a specific ApplReg event.

RowSource Event

The RowSource event logs the number of rows processed by a single row source within an execution plan. Items associated with the RowSource event are shown in Table 12–13. The name of the formatter table is V_192216243_F_5_E_6_7_3.

Table 12–13 Items Associated with the RowSource Event

Session_Index	Session_Serial	Cursor_Number
Position_ID	Row_Count	Cross-Product Items 1-5

The combination of Session_Index, Session_Serial, Cursor_Number, and Position_ID uniquely identifies a RowSource event.

RowSource Event (Specific to Oracle Server Release 8.0.2 and Higher)

The RowSource event logs the number of rows processed by a single row source within an execution plan. Items associated with the RowSource event for Oracle Server release 8.0.2 or higher are shown in Table 12–14. The name of the formatter table is V_192216243_F_5_E_6_8_0.

Table 12–14 Items Associated with the RowSource Event

Session_Index	Session_Serial	Cursor_Number
Operation_ID	Row_Count	Parent_Op_ID
Position	Object_ID	Operation
Cross-Product Items 1-6		

The combination of Session_Index, Session_Serial, Cursor_Number, and Operation_ID uniquely identifies a RowSource event.

Note: The text in the Operation item is equivalent to information about the execution plan, which is similar to data that can be obtained by running explain plan.

SQLSegment Event

The SQLSegment event is a description of a SQL statement. Items associated with the SQLSegment event are shown in Table 12–15. The name of the formatter table is V_192216243_F_5_E_7_7_3.

Table 12–15 Items Associated with the SQLSegment Event

Session_Index	Session_Serial	Cursor_Number
SQL_Text_Hash	Lib_Cache_Addr	SQL_Text_Segment
SQL_Text	Cross-Product Items 1-6	

A SQL segment does not have an explicit identifier. The SQL_Text_Hash field is always the same for each occurrence of a SQL statement but multiple statements can have the same hash value. If a statement is forced out of the library cache and then swapped back in, the same statement can have multiple values for Lib_Cache_Addr. The combination of Session_Index, Session_Serial, SQL_Text_Hash, and Lib_Cache_Addr usually should identify a particular SQL statement for a session. If you add Cursor_Number, you identify a particular occurrence of a SQL statement within the session.

Wait Event

The wait event shows the total waiting time in hundredths of seconds for all responses. Items associated with the wait event are shown in Table 12–16. The name of the formatter table is V_192216243_F_5_E_13_7_3.

Table 12–16 Items Associated with the Wait Event

Session_Index	Session_Serial	Wait_Time
P1	P2	P3
Description	Cross-Product Items 1-7	

The wait event does not have an explicit identifier. The combination of Session_Index, Session_Serial, Description, Timestamp, and Timestamp_Nano should uniquely identify a specific wait event.

Parse Event

The Parse event records the start and end of the parsing phase during the processing of a SQL statement. The parsing phase occurs when the SQL text is read

in and broken down (parsed) into its various components. Tables and fields are identified, as well as which fields are sort criteria and which information needs to be returned. Items associated with the parse event are shown in Table 12–17. The name of the formatter table is V_192216243_F_5_E_8_7_3.

Table 12–17 Items Associated with the Parse Event

Items for Start of Parse Event		
Session_Index	Session_Serial	Event Statistics Block
Cursor_Number	Resource Items	Cross-Product Items 1-6
Items for End of Parse Event		
Session_Index	Session_Serial	Event Statistics Block
Cursor_Number	Depth	Missed
Oracle_Cmd_Type	Optimizer_Mode	Current_UID
Schema_UID	SQL_Text_Hash	Lib_Cache_Addr
Resource Items		

The combination of Session_Index, Session_Serial, Cursor_Number, and SQL_Text_Hash uniquely identifies a specific parse event.

Execute Event

The Execute event is where the query plan is executed. That is, the parsed input is analyzed to determine exact access methods for retrieving the data, and the data is prepared for fetch if necessary. Items associated with the Execute event are shown in Table 12–18. The name of the formatter table is V_192216243_F_5_E_9_7_3.

Table 12–18 Items Associated with the Execute Event

Items for Start of Execute Event		
Session_Index	Session_Serial	Event Statistics Block
Cursor_Number	Resource Items	Cross-Product Items 1-6
Items for End of Execute Event		
Session_Index	Session_Serial	Event Statistics Block
Cursor_Number	Depth	Missed
Row_Count	SQL_Text_Hash	Lib_Cache_Addr
Resource Items		

The combination of Session_Index, Session_Serial, Cursor_Number, and SQL_Text_Hash uniquely identifies a specific Execute event.

Fetch Event

The Fetch event is the actual return of the data. Multiple fetches can be performed for the same statement to retrieve all the data. Items associated with the Fetch event are shown in Table 12–19. The name of the formatter table is V_192216243_F_5_E_10_7_3.

Table 12–19 Items Associated with the Fetch Event

Items for Start of Fetch Event		
Session_Index	Session_Serial	Event Statistics Block
Cursor_Number	Resource Items	Cross-Product Items 1-6
Items for End of Fetch Event		
Session_Index	Session_Serial	Event Statistics Block
Cursor_Number	Depth	Row_Count
End_of_Fetch	SQL_Text_Hash	Lib_Cache_Addr
Resource Items		

The combination of Session_Index, Session_Serial, Cursor_Number, SQL_Text_Hash, Timestamp, and Timestamp_Nano uniquely identifies a specific Fetch event.

LogicalTX Event

The LogicalTX event logs the start and end of a logical transaction (that is, statements issued that may cause a change to the database status). Items associated with the LogicalTX event are shown in Table 12–20. The name of the formatter table is V_192216243_F_5_E_11_7_3.

Table 12–20 Items Associated with the LogicalTX Event

Items for Start of LogicalTX Event		
Session_Index	Session_Serial	Event Statistics Block
TX_Type	TX_SO_Addr	Resource Items
Cross-Product Items 1-6		
Items for End of LogicalTX Event		
Session_Index	Session_Serial	Event Statistics Block
TX_Type	TX_SO_Addr	Resource Items

The transaction identifier stored in CROSS_FAC_4 should uniquely identify a specific transaction. Or, use Session_Index, Session_Serial, and TX_SO_Addr.

PhysicalTX Event

The PhysicalTX event logs the start and end of a physical transaction (that is, statements issued that caused a change in database status). Items associated with the PhysicalTX event are shown in Table 12–21. The name of the formatter table is V_192216243_F_5_E_12_7_3.

Table 12–21 Items Associated with the PhysicalTX Event

Items for Start of PhysicalTX Event		
Session_Index	Session_Serial	Event Statistics Block
TX_Type	TX_ID	Resource Items
Cross-Product Items 1-6		
Items for End of PhysicalTX Event		
Session_Index	Session_Serial	Event Statistics Block
TX_Type	TX_ID	Commit_Abort
Resource Items		

The transaction identifier stored in CROSS_FAC_4 should uniquely identify a specific transaction.

Event Set File Names

Oracle Trace events can be organized into **event sets** that restrict the data collection to specific events. You can establish event sets for performance monitoring, auditing, diagnostics, or any logical event grouping.

Table 12–22 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.
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).
SQLSTATS	SQL_AND_STATS event set. Collects SQL text and statistics only.

Table 12–22 Server Event Set File Names (Cont.)

Event Set File Name (.fdf)	Description
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).
SQLWAITS	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.

See Also: "Event Sets" on page 12-2

Troubleshooting Oracle Trace

Use the following sections to troubleshoot problems while using Oracle Trace.

Oracle Trace Configuration

If you suspect an Oracle Trace configuration problem:

- Examine the EPC_ERROR.LOG file for details of any logged Oracle Trace errors.
- Look for the administration files on $ORACLE_HOME/otrace/admin (*.dat files) and run otrccref to recreate the Oracle Trace *.dat files if the files do not exist.
- Verify that the .fdf files are in the $ORACLE_HOME/otrace/admin/fdf directory.
- Verify the correct version of Oracle Trace Collection Services match the appropriate Oracle Server version

```
% $ORACLE_HOME/bin/otrccol version
```

Table 12–23 Matching Releases of Oracle Trace Collection Services with Releases of Oracle Server

If the Returned Value Is:	Then the Command-Line Interface Release Is:
1	733
2	803
3	734
4	804
5	805
6	813
7	814
8	815
9	806
10	816
11	817
12	901

- To verify that a collection is currently running, use the command-line interface to check the status:

  ```
  % $ORACLE_HOME/bin/otrccol check <collection_name>
  ```

To test the CLI:

1. CLI needs to run from a privileged account, for example, the Oracle operating system user account.
2. The Oracle home and SID environmental variables must be set properly.

   To check settings on UNIX:

   ```
   printenv ORACLE_HOME
   printenv ORACLE_SID
   ```

   To set settings on UNIX:

   ```
   setenv ORACLE_HOME <path>
   setenv ORACLE_SID <sid>
   ```

There should be one CLI per ORACLE_HOME. For example, if you have two Oracle Server release 7.3.3 instances sharing the same ORACLE_HOME, there should be only one CLI.

3. Verify that the collection name has not already been used before you start the collection.

   Look for <*collection name*>.cdf and .dat files in:

   - $ORACLE_HOME/otrace/admin/cdf directory
   - The directory specified in database parameter ORACLE_TRACE_COLLECTION_PATH
   - The directory specified by EPC_COLLECTION_PATH environment variable

4. If you want to generate database activity for this collection, connect to the database.

   - For Oracle Server release 7.3.x, connect to the service before you create your collection.
   - For Oracle Server release 8.0, you can connect to the database anytime and the processes are registered.

Server Environment

If you suspect a server environment problem, verify the following:

- The server node has sufficient disk space for the collection output files. If there is not sufficient disk space, the collection stops. To limit the size of collections, use the Trace option for limiting collection size. For a description on how to limit collection size, see the Using the Oracle Trace CLI section in this chapter.

  To solve the immediate problem, stop the collection, and free up space so Oracle Trace can end the collection.

  Initially limiting the collection to specific users and/or wait events also helps to limit the amount of data collected. Limiting users and wait events is available for Oracle Server releases 8.0.4 and higher.

- Your session does not participate in more than five collections. Sessions log data for the five most recent collections. Thus, if you have more than five collections, data is missing for the oldest collection.

Missing Data

Wait Times Were Not Collected Wait times are collected only if the INIT<*sid*>.ORA parameter, TIMED_STATISTICS, is set to true.

Missing SQL Statement from Collection If an expected SQL statement does not appear to be in your collection, it may be because a small amount of data in the Oracle Trace data collection buffers may not have been flushed out to the collection data file, even though the collection has been stopped. Additional database activity should flush these buffers to disk, and shutting down the database also forces a flush of these buffers.

Collection Is Too Large There may be times when a collection is too large. Starting with Oracle Server release 8.0.4, you can collect data for specific users and specific wait event types to minimize the size of the collection. Because, almost always, the server is waiting for a latch, lock, or resource, wait event data for a brief collection can be quite extensive.

Collection Is Empty In Oracle8 databases (prior to Oracle 8.1.7), the ORACLE_TRACE_ENABLE parameter in the INIT*sid*.ORA file on the server must be set to true before the database is started. Starting with Oracle 8.1.7 it is dynamic and may be modified via ALTER_SESSION or ALTER_SYSTEM. (For Oracle7 the ORACLE_TRACE_ENABLE parameter should be left as false, unless you are using the init.ora parameter method to start or stop collections.) You can also see this problem if there are too many collections running concurrently.

Oracle Trace Could Not Access Memory

On Windows NT systems, if you are running Oracle Trace collections and an error occurs indicating Oracle Trace could not access memory, the collect.dat file has become full. You must create a new .dat file by running the otrccref.exe image located in the $ORACLE_HOME/bin directory. However, database services must be shutdown to release the collect.dat file for the otrccref script to be able to create the new collect.dat file. You can also increase the number of collect.dat records above the default of 36 records (for example, otrccref -c50 to create a new otrace/admin/collect.dat file containing 50 records).

Oracle7 Stored Procedures

If the attempt to collect Oracle Trace data for an Oracle7 database results in the message "Error starting / stopping Oracle7 database collection," this may be due to

missing database stored procedures that Oracle Trace uses to start and stop Oracle7 collections.

- Check for stored procedures (for Oracle Server releases 7.3.x)

  To check for stored procedures using the Oracle Enterprise Manager console, use the Navigator and the following path:

  ```
  Networks=>Databases=><your database>=>Schema Objects=>Packages=>SYS
  ```

  Look for stored procedures starting with DBMS_ORACLE_TRACE_xxx.

  To check for stored procedures using Oracle Server Manager or Oracle SQL*Plus Worksheet:

  ```
  select object_name from dba_objects where object_name like '%TRACE%'
  and object_type = 'PACKAGE';
  OBJECT_NAME
  DBMS_ORACLE_TRACE_AGENT
  DBMS_ORACLE_TRACE_USER
  2 rows selected.
  ```

For Oracle7, Oracle Trace required that these stored procedures be installed on the database. These SQL scripts may be automatically run during database installation depending on the platform-specific installation procedures. If they are not executed during database installation, you must run these scripts manually. You can add these stored procedures to the database by running the otrcsvr.sql script from $ORACLE_HOME/otrace/admin) from a privileged database account (SYS or INTERNAL). To run the script, set the default to the path were the script is located. This script runs other scripts that do not have the path specified. These other scripts fail if you are not in the directory where these scripts will run.

EPC_ERROR.LOG File

The EPC_ERROR.LOG file provides information about the collection processing, specifically the Oracle Trace Collection Services errors.

The EPC_ERROR.LOG file is created in the current default directory.

For general information about causes and actions for most Oracle Trace messages, see the *Oracle Enterprise Manager Messages Manual*.

Formatter Tables

Oracle Server releases 7.3.4 and 8.0.4 and later automatically create the formatter tables. Prior to Oracle Server releases 7.3.4 and 8.0.4, you must run the

`otrcfmtc.sql` script from Oracle Server Manager or Oracle SQL*Plus Worksheet as the user who will be formatting the data. If you must manually execute `otrcfmtc.sql` to create the formatter tables, use the SQL script from the same Oracle home as your collections to be formatted.

The `otrcfmtc.sql` script is located in the `$ORACLE_HOME/otrace/admin` directory.

Formatting error might be due to one of the following causes:

1. The user did not run the script to create the formatter tables (valid for releases of Oracle Server prior to 7.3.4 and 8.0.4).
2. The formatter tables were not created by the same user ID that was used when the collection was created (valid for releases of Oracle Server prior to 7.3.4 and 8.0.4).

Look for `EPC_COLLECTION`.

To check for formatter tables using SQL Worksheet:

```
CONNECT <username>/<password>@<service name>
DESCRIBE epc_collection
```

Part III

Creating a Database for Good Performance

Part III describes how to configure a database for good performance.

The chapters in this part are:

- Chapter 13, "Building a Database for Performance"
- Chapter 14, "Memory Configuration and Use"
- Chapter 15, "I/O Configuration and Design"
- Chapter 16, "Understanding Operating System Resources"
- Chapter 17, "Configuring Instance Recovery Performance"
- Chapter 18, "Configuring Undo and Temporary Segments"
- Chapter 19, "Configuring Shared Servers"

13

Building a Database for Performance

One of the first stages in managing a database is the initial database creation. Although performance modifications can be made to both the database and to the Oracle instance at a later time, much can be gained by carefully designing the database for the intended needs. This chapter contains the following sections:

- Initial Database Creation
- Creating Tables for Good Performance
- Loading and Indexing Data
- Initial Instance Configuration
- Setting up OS, Database, and Network Monitoring

Note: This chapter is an overview of. Before reading this chapter, it may be very valuable to read the information in the *Oracle9i Database Performance Methods* manual. For detailed information on memory and I/O configuration, see the other chapters in this part.

Initial Database Creation

Database Creation using the Installer

The Oracle Installer lets you create a database during software installation or at a later time using the Database Creation Assistant. This is an efficient way of creating databases for small to medium size environments, and it provides a straightforward graphical user interface. However, this procedure sets some limits on the possibilities for various options, and it is therefore not recommended for database creation in larger environments.

Manual Database Creation

A manual approach provides full freedom for different configuration options. This is especially important for larger environments. A manual approach typically involves designing multiple parameter files for use during the various stages, running SQL scripts for the initial CREATE DATABASE and subsequent CREATE TABLESPACE statements, running necessary data dictionary scripts, and so on.

Parameters Necessary for Initial Database Creation

The initialization parameter file is read whenever an Oracle instance is started, including the very first start before the database is created. Very few parameters must be set before the initial database creation, because they cannot be modified at a later time. These parameters are:

DB_BLOCK_SIZE	This sets the size of the Oracle database blocks stored in the database files and cached in the SGA. The range of values depends on the operating system, but it is typically powers of two in the range 2048 to 16384. Common values are 4096 or 8192 for transaction processing systems and higher values for database warehouse systems.
DB_NAME DB_DOMAIN	These set the name of the database and the domain name of the database respectively. Although they can be changed at a later time, it is highly advisable to set these correctly before the initial creation. The names chosen must be reflected in the SQL*Net configuration as well.
COMPATIBLE	This specifies the release with which the Oracle server must maintain compatibility. It allows you to take advantage of the maintenance improvements of a new release immediately in your production systems without testing the new functionality in your environment. If your application was designed for a specific release of Oracle, and you are actually installing a later release, then you might want to set this parameter to the version of the previous release.

See Also:

- Chapter 15, "I/O Configuration and Design" and *Oracle9i Database Performance Methods* for more information on the efficient use of datafiles in both the initial database creation and subsequent tablespace creations
- *Oracle9i Database Reference* for more information on initialization parameters

The CREATE DATABASE Statement

The first SQL statement that is executed after startup of the initial instance is the CREATE DATABASE statement. This creates the initial system tablespace, creates the initial redo logfiles, and sets certain database options. The following options cannot be changed or can only be changed with difficulty at a later time:

Character set	The character set specified by this option identifies the character set used for SQL text, for the internal data dictionary, and most importantly for text stored as datatypes CHAR, VARCHAR, or VARCHAR2. After data including any type of national characters has been loaded, the character set cannot be changed.
National character set	This character set is used for the datatypes NCHAR, NVARCHAR, and NVARCHAR2. In general, as with the regular character set, it cannot be changed.
SQL.BSQ file	This creates the internal data dictionary. For information on modifying this file, see Chapter 15, "I/O Configuration and Design".
Location of initial datafile	The initial datafile(s) that will make up the system tablespace should be set with care. They can be modified later, but the procedure involves a complete shutdown of the instance.

Example of a CREATE DATABASE Script

```
CONNECT SYS/ORACLE AS SYSDBA
STARTUP NOMOUNT pfile=/u01/admin/init_create.ora'
CREATE DATABASE "dbname"
DATAFILE '/u01/oradata/system01.dbf' size 200M
LOGFILE  '/u02/oradata/redo01.dbf' size 100M,
         '/u02/oradata/redo02.dbf' size 100M
CHARACTER SET "WE8ISO8859P1"
NATIONAL CHARACTER SET "UTF8";
```

Running Data Dictionary Scripts

After running the CREATE DATABASE statement, certain catalog scripts must be executed. They are found in the RDBMS/ADMIN directory on UNIX or the RDBMS\ADMIN directory on Windows, under the ORACLE_HOME directory. The following scripts must be executed.

CATALOG.SQL	Needed for all normal data dictionary views
CATPROC.SQL	Needed to load the initial PL/SQL environment

Note: When specific options or features are in use (for example, Java or replication), more scripts are necessary. These are documented with each individual option.

Example of Executing Required Data Dictionary Scripts

```
CONNECT SYS/ORACLE AS SYSDBA
@@CATALOG
@@CATPROC
```

The use of the double at-sign forces execution of these scripts from the proper directory.

Sizing Redo Log Files

The size of the redo log files can influence performance, because the behavior of the database writer and archiver processes depend on the redo log sizes. Generally, larger redo log files provide better performance. Small log files can increase checkpoint activity and reduce performance. Because the recommendation on I/O distribution for high performance is to use separate disks for the redo log files, there is no reason not to make them large. A potential problem with large redo log files is that these are a single point of failure if redo log mirroring is not in effect.

It is not possible to provide a specific size recommendation for redo log files, but redo log files in the range of a hundred megabytes to a few gigabytes are considered reasonable. Size your online redo log files according to the amount of redo your system generates. A rough guide is to switch logs at most once every twenty minutes.

The complete set of required redo log files can be created during database creation. After they are created, the size of a redo log size cannot be changed. However, new, larger files can be added later, and the original (smaller) ones can subsequently be dropped.

Not much can be done to speed up the creation of the initial database and the loading of necessary data dictionary views from catalog SQL files. These steps must be run serially after each other.

Note: Although the size of the redo log files does not affect LGWR performance, it can affect DBWR and checkpoint behavior.

Creating Subsequent Tablespaces

After creating the initial database, several extra tablespaces must be created. All databases should have at least three tablespaces in addition to the system tablespace: a temporary tablespace, which is used for things like sorting; a rollback tablespace, which is used to store rollback segments or is designated as the automatic undo management segment; and at least one tablespace for actual application use. In most cases, applications require several tablespaces. For extremely large tablespaces with many datafiles, multiple ALTER TABLESPACE *X* ADD DATAFILE *Y* statements can also be run in parallel.

During tablespace creation, the datafiles that make up the tablespace are initialized with zero values. Oracle does this to ensure that all datafiles can be written in their entirety, but this can obviously be a lengthy process if done in serial. Therefore, run multiple CREATE TABLESPACE statements concurrently to speed up the tablespace creation process. The most efficient way to do this is to run one SQL script for each set of available disks.

For permanent tables, the choice between local and global extent management on tablespace creation can have a large effect on performance. For any permanent tablespace that has moderate to large insert, modify, or delete operations compared to reads, local extent management should be chosen.

Note: Tablespaces created as temporary cannot use local extent management.

Examples of How to Concurrently Create Tablespaces

```
CONNECT SYSTEM/MANAGER
CREATE TABLESPACE appdata DATAFILE
  '/u02/oradata/appdata01.dbf' size 1000M;
```

In another session:

```
CONNECT SYSTEM/MANAGER
CREATE TABLESPACE appindex DATAFILE
  '/u03/oradata/appindex01.dbf' size 1000M;
```

Creating Tables for Good Performance

When you create a segment, such as a table, Oracle allocates space in the database for the data. If subsequent database operations cause the data volume to increase and exceed the space allocated, then Oracle extends the segment.

When installing applications, an initial step is to create all necessary tables and indexes. This operation is by itself relatively fast, and not much is gained by doing it in parallel. However, some things require attention:

- Use Locally Managed Tablespaces: If the tablespace which contains the segment is dictionary managed, and there is a significant amount of dynamic extension, this can affect performance. Store such tables in locally managed tablespaces, and determine appropriate extent sizes.
- Choosing Extent Sizes: The extent sizes for segments should be N orders of magnitude greater than the DB_FILE_MULTIBLOCK_READ_COUNT. This ensures that mutliblock reads can be performed effectively.
- Setting Storage Options: Applications should carefully set storage options for the intended use of the table or index. This includes setting the values for PCTFREE and PCTUSED. (Using automatic segment-space management eliminates the need to specify PCTUSED).
- Using Freelists: For objects with high insert activity, freelists can increase scalability. However, the freelist settings must be done when the table is created. After data is inserted, the table cannot be altered to use freelists.

  Note: Freelists have been the traditional method of managing free space within segments. Automatic segment-space management provides a simpler and more efficient way of managing segment space, and completely eliminates any need to specify and tune the PCTUSED, FREELISTS, and FREELISTS GROUPS attributes for segments. If such attributes are specified, they are ignored.

  To use automatic segment-space management, create locally managed tablespaces, with the segment space management clause set to AUTO. For more information on creating and using automatic segment-space management, see *Oracle9i Database Administrator's Guide*.

- Setting INITRANS: Each datablock has a number of transaction entries that are used for row locking purposes. Initially, this number is specified by the INITRANS parameter, and the default value (1 for tables, 2 for indexes) is generally sufficient. However, if a table or index is known to have many rows per block with a high possibility of many concurrent updates, then it is beneficial to set a higher value. This must be done at the CREATE TABLE/CREATE INDEX time to ensure that it is set for all blocks of the object.

Loading and Indexing Data

Many applications need to load data as part of the initial application installation process. This can be fixed data, such as postal code or other type of lookup data, or it can be actual data originating in older systems. Oracle's SQL*Loader tool is the most efficient way to load a substantial amount of data.

Using SQL*Loader for Good Performance

When running SQL*Loader, you specify to use either the conventional path or the direct path. The conventional path uses ordinary SQL `INSERT` statements to load data into the tables, which means that the loading can be done concurrently with other database operations. However, the loading then is also limited by the normal `INSERT` performance. For quick loading of large amounts of data, choose the direct path. With the direct path, the loading process bypasses SQL and loads directly into the database blocks. During this type of load, normal operation on the table (or partition for partitioned tables) cannot be performed.

The following tips could help speed up the data loading process using SQL*Loader:

- Use fixed length fields rather than delimited or quoted fields. This reduces the time needed for SQL*Loader to read and interpret the input file.
- Run SQL*Loader locally rather than via a network connection.
- Load numbers read from the input file as text rather than in binary format. This might be faster, especially if the numbers are loaded without any computation. This is true for integers and for floating point numbers.
- For very large load operations, use SQL*Loader to do parallel data load.

> **See Also:** *Oracle9i Database Utilities* for detailed information on SQL*Loader

Efficient Index Creation

The most efficient way to create indexes is to create them after data has been loaded. By doing this, space management becomes much simpler, and no index maintenance takes place for each row inserted. SQL*Loader automatically does this, but if you are using other methods to do initial data load, you might need to do this manually. Additionally, index creation can be done in parallel using the `PARALLEL` clause of the `CREATE INDEX` statement. However, SQL*Loader is not able to do this, so you must manually create indexes in parallel after loading data.

During index creation on tables that contain data, the data must be sorted. This sorting is done in the fastest possible way, if all available memory is used for sorting. This can be controlled with the startup parameter SORT_AREA_SIZE. The value of this parameter should be set using the following rules:

1. Find the amount of available memory by subtracting the size of the SGA and the size of the operating system from the total system memory.
2. Divide this amount by the number of parallel slaves that you will use; this is typically the same as the number of CPUs.
3. Subtract a process overhead, typically a five to ten megabytes, to get the value for SORT_AREA_SIZE.

Note: You can also save time on index creating operations (or fast rebuilds) with on the fly statistics generation.

Example of Creating Indexes Efficiently

A system with 512 Mb memory runs an Oracle instance with a 100 Mb SGA, and the operating system uses 50 Mb. Hence, the memory available for sorting is 362 Mb (512 minus 50 minus 100). If the system has four CPUs and you run with four parallel slaves, then each of these will have 90 Mb available, 10 Mb are set aside for process overhead, and SORT_AREA_SIZE should be set to 80 Mb. This can be done either in the initialization file or on a per session basis with a statement like the following:

```
ALTER SESSION SET SORT_AREA_SIZE = 80000000;
```

Initial Instance Configuration

A running Oracle instance is configured using startup parameters, which are set in the initialization parameter file. These parameters influence the behavior of the running instance, including influencing performance. In general, a very simple initialization file with few relevant settings covers most situations, and the initialization file should not be the first place you expect to do performance tuning, except for the very few parameters shown below.

See Also: Chapter 14, "Memory Configuration and Use"

The following describes the parameters necessary in a minimal initialization file. Although these parameters are necessary, but they have no performance impact:

DB_NAME	Name of the database. This should match the ORACLE_SID environment variable.
DB_DOMAIN	Location of the database in Internet dot notation.
OPEN_CURSORS	Limit on the maximum number of cursors (active SQL statements) per session. The setting is application-dependent, and the default, in many cases, is sufficient.
CONTROL_FILES	Set to contain at least two files on different disk drives to prevent failures from control file loss.
DB_FILES	Set to the maximum number of files that can assigned to the database.

The following list includes the most important parameters to set with performance implications:

DB_BLOCK_SIZE	Sets the database block size.
DB_BLOCK_ BUFFERS	Size of the buffer cache in the SGA. There are no good and simple rules to set a value, which is very application dependent, but typical values are in the range of twenty to fifty per user session. More often, this value is set too high than too low.
SHARED_POOL_ SIZE	Sets the size of the shared pool in the SGA. The setting is application-dependent, but it is typically is in the range of a few to a few tens of megabytes per user session.
PROCESSES	Sets the maximum number of processes that can be started by that instance. This is the most important primary parameter to set, because many other parameter values are deduced from this.
SESSIONS	This is set by default from the value of processes. However, if you are using the shared server, then the deduced value is likely to be insufficient.
JAVA_POOL_SIZE	If you are using Java stored procedures, then this parameter should be set depending on the actual requirements of memory for the Java environment.
LOG_ARCHIVE_ XXX	Enables redo log archiving. See *Oracle9i User-Managed Backup and Recovery Guide*.

ROLLBACK_ SEGMENTS	Allocates one or more rollback segments by name to this instance. If you set this parameter, the instance acquires all of the rollback segments named in this parameter, even if the number of rollback segments exceeds the minimum number required by the instance (calculated as TRANSACTIONS /TRANSACTIONS_PER_ROLLBACK_SEGMENT).

Example of a Minimal Initialization File

In many cases, only the parameters mentioned below need to be set to appropriate values for the Oracle instance to be reasonable well-tuned. Below is an example of such an initialization file:

```
DB_NAME = finance
DB_DOMAIN = hq.company.com
CONTROL_FILES = ('/u01/database/control1.dbf', '/u02/database/control2.dbf')
DB_BLOCK_SIZE = 8192
DB_BLOCK_BUFFERS = 12000 # this is approximately 100 Mb
DB_FILES = 200 # Maximum 200 files in the database
SHARED_POOL_SIZE = 100000000 # 100 Mb
PROCESSES = 80 # Would be good for approximately 70
# directly connected users
# log_archive_XXX
# Set various archiving parameters
```

Configuring Rollback Segments

Oracle needs rollback space to keep information for read consistency, for recovery, and for actual rollback statements. This is kept either in rollback segments or in one or more automatic undo management tablespaces.

The V$UNDOSTAT view contains statistics for monitoring and tuning undo space. Using this view, you can better estimate the amount of undo space required for the current workload. Oracle also uses this information to help tune undo usage in the system. The V$ROLLSTAT view contains information about the behavior of the undo segments in the undo tablespace.

See Also:

- *Oracle9i Database Administrator's Guide* for detailed information on managing undo space using rollback segments or using automatic undo management
- Chapter 24, "Dynamic Performance Views for Tuning" for more information on the V$UNDOSTAT and V$ROLLBACK views

Setting up OS, Database, and Network Monitoring

To effectively diagnose performance problems, it is vital to have an established performance baseline for later comparison when the system is running poorly. Without a baseline data point, it can be very difficult to identify new problems. For example, perhaps the volume of transactions on the system has increased, or the transaction profile or application has changed, or the number of users has increased.

After the database is created, tables are created, data is loaded and indexed, and the instance is configured, it is time to set up monitoring procedures.

See Also: Chapter 20, "Oracle Tools to Gather Database Statistics"

14

Memory Configuration and Use

This chapter explains how to allocate memory to Oracle memory caches, and how to use those caches. Proper sizing and effective use of the Oracle memory caches greatly improves database performance.

This chapter contains the following sections:

- Understanding Memory Allocation Issues
- Configuring and Using the Buffer Cache
- Configuring and Using the Shared Pool and Large Pool
- Configuring and Using the Java Pool
- Configuring and Using the Redo Log Buffer
- Configuring the PGA Working Memory
- Reducing Total Memory Usage

Understanding Memory Allocation Issues

Oracle stores information in memory caches and on disk. Memory access is much faster than disk access. Physical I/O takes a significant amount of time when compared with memory access, typically in the order of ten milliseconds. Physical I/O also increases the CPU resources required, because of the path length in device drivers and operating system event schedulers. For this reason, it is more efficient for data requests for frequently accessed objects to be satisfied solely by memory, rather than also requiring disk access.

The performance goal is to reduce the physical I/O overhead as much as possible by making it more likely that the required data is in memory, or by making the process of retrieving the required data more efficient.

Oracle Memory Caches

The main Oracle memory caches that affect performance are the following:

- Shared pool
- Log buffer
- Buffer cache
- Process-private memory (for example, used for sorting, hashing, and so on)

The size of these memory caches is configurable using initialization configuration parameters. The values for these parameters are also dynamically configurable using the `ALTER SYSTEM` statement (except for the log buffer, which is static after startup).

Dynamically Changing Cache Sizes

It is possible to reconfigure the sizes of the shared pool and the buffer cache dynamically, in addition to dynamically reconfiguring process-private memory. Memory for the shared pool and buffer cache is allocated in units of granules. A granule can be 4 MB or 16 MB, depending on the total size of your SGA at the time of instance startup. If the size of your SGA is less than 128 MB, then the granules are 4 MB in size; otherwise, they are 16 MB. It is possible to decrease the size of one cache and reallocate that memory to another cache, if needed. The total SGA size can be expanded to a value equal to the `SGA_MAX_SIZE` parameter.

Note: `SGA_MAX_SIZE` cannot be dynamically resized.

The maximum amount of memory usable by the instance is determined at instance startup by the initialization parameter SGA_MAX_SIZE. You can specify SGA_MAX_SIZE to be larger than the sum of all of the memory components (such as buffer cache and shared pool); otherwise, SGA_MAX_SIZE defaults to the actual size used by those components. Setting SGA_MAX_SIZE larger than the sum memory used by all of the components lets you dynamically increase a cache size without needing to decrease the size of another cache.

> **See Also:** *Oracle9i Database Concepts* or *Oracle9i SQL Reference* for more information on the dynamic SGA

Application Considerations

With memory configuration, it is important to size the cache appropriately for the application's needs. Conversely, tuning the application's use of the caches can greatly reduce the resource requirements. Efficient use of the Oracle memory caches also reduces the load on other related resources, such as the latches that protect those caches, the CPU, and the I/O system.

For best performance, consider the following:

- The application should be designed and coded to interact with Oracle efficiently.
- Memory allocations to Oracle memory structures should best reflect the needs of the application.

Making changes or additions to an existing application might require resizing Oracle memory structures to meet the needs of your modified application.

Operating System Memory Use

For most operating systems, it is important to consider the following:

- Reduce Paging

  Paging is when an OS transfers memory-resident pages disk solely to allow new pages to be loaded into memory. Many operating systems page to accommodate large amounts of information that do not fit into real memory. On most operating systems, paging reduces performance.

  Examine the operating system using OS utilities to identify whether there is a lot of paging. If so, then the total memory on the system might 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.

- Fit 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 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 paging significantly outweighs the advantage of a large SGA.

  To see how much memory is allocated to the SGA and each of its internal structures, enter the following SQL*Plus statement:

  ```
  SHOW SGA
  ```

  The output of this statement will look similar to the following:

  ```
  Total System Global Area  840205000 bytes
  Fixed Size                  279240 bytes
  Variable Size             520093696 bytes
  Database Buffers          318767104 bytes
  Redo Buffers                1064960 bytes
  ```

- Allow Adequate Memory to Individual Users

  When sizing the SGA, ensure that you allow enough memory for the individual server processes and any other programs running on your system.

  > **See Also:** Your operating system hardware and software documentation, as well as your Oracle operating system-specific documentation, for more information on tuning operating system memory usage

Iteration During Configuration

Configuring memory allocation involves distributing available memory to Oracle memory structures, depending on the needs of your application. The distribution of memory to Oracle structures can affect the amount of physical I/O necessary for Oracle to operate. Having a good first initial memory configuration also provides an indication of whether the I/O system is effectively configured.

It might be necessary to repeat the steps of memory allocation after the initial pass through the process. Subsequent passes let you make adjustments in earlier steps based on changes in later steps. For example, decreasing the size of the buffer cache

allows you to increase the size of another memory structure, such as the shared pool.

Configuring and Using the Buffer Cache

For many types of operations, Oracle uses the buffer cache to cache blocks read from disk. Oracle bypasses the buffer cache for particular operations, such as sorting and parallel reads. For operations that use the buffer cache, this section explains the following:

- Using the Buffer Cache Effectively
- Sizing the Buffer Cache
- Interpreting and Using the Buffer Cache Statistics
- Considering Multiple Buffer Pools

Using the Buffer Cache Effectively

To use the buffer cache effectively, the application's SQL statements should be tuned to avoid unnecessary resource consumption. To ensure this, verify that frequently executed SQL statements and SQL statements that perform many buffer gets have been tuned.

> **See Also:** Chapter 6, "Optimizing SQL Statements" for information on how to do this

Sizing the Buffer Cache

When configuring a brand new instance, it is impossible to know the correct size for the buffer cache. Typically, a DBA makes a first estimate for the cache size, then runs a representative workload on the instance and examines the relevant statistics to see whether the cache is under- or over-configured.

Buffer Cache Statistics

There are a number of different statistics that can be used to examine buffer cache activity. These include the following:

- `V$DB_CACHE_ADVICE`
- Buffer cache hit ratio

Using V$DB_CACHE_ADVICE This view is populated when the DB_CACHE_ADVICE parameter is set to ON. This view shows the estimated miss rates for twenty potential buffer cache sizes, ranging from 10% of the current size to 200% of the current size. Each of the twenty potential cache sizes has its own row in this view, with the predicted physical I/O activity that would take place for that cache size. The DB_CACHE_ADVICE parameter is dynamic, so the advisory can be enabled and disabled dynamically to allow you to collect advisory data for a specific workload.

There are two minor overheads associated with this advisory:

- CPU: When the advisory is enabled, there is a small increase in CPU usage, because additional bookkeeping is required.
- Memory: The advisory requires memory to be allocated from the shared pool (of the order of 100 bytes per buffer). This memory is preallocated on instance startup if DB_CACHE_ADVICE is set to READY in anticipation of collecting advisory statistics, or if the parameter is set to ON. If the parameter is set to OFF (the default setting) on instance startup, then the memory is dynamically allocated from the shared pool at the time the parameter value is modified to a value other than OFF.

The parameter DB_CACHE_ADVICE should be set to ON, and a representative workload should be running on the instance. Allow the workload to stabilize before querying V$DB_CACHE_ADVICE view.

The following SQL statement returns the predicted I/O requirement for the default buffer pool for various cache sizes:

```
column size_for_estimate         format 999,999,999,999 heading 'Cache Size (m)'
column buffers_for_estimate      format 999,999,999 heading 'Buffers'
column estd_physical_read_factor format 999.90 heading 'Estd Phys|Read Factor'
column estd_physical_reads       format 999,999,999 heading 'Estd Phys| Reads'

SELECT size_for_estimate, buffers_for_estimate
     , estd_physical_read_factor, estd_physical_reads
  FROM V$DB_CACHE_ADVICE
 WHERE name          = 'DEFAULT'
   AND block_size    = (SELECT value FROM V$PARAMETER
                        WHERE name = 'db_block_size')
   AND advice_status = 'ON';
```

The following output shows that if the cache was 212MB, rather than the current size of 304MB, the estimated additional number of physical reads would be 17 million (17,850,847). Increasing the cache size beyond it's current size would not provide a significant benefit.

Cache Size (MB)	Buffers	Estd Phys Read Factor	Estd Phys Reads	
30	3,802	18.70	192,317,943	**10% of Current Size**
60	7,604	12.83	131,949,536	
91	11,406	7.38	75,865,861	
121	15,208	4.97	51,111,658	
152	19,010	3.64	37,460,786	
182	22,812	2.50	25,668,196	
212	26,614	1.74	17,850,847	
243	30,416	1.33	13,720,149	
273	34,218	1.13	11,583,180	
304	38,020	1.00	10,282,475	**Current Size**
334	41,822	.93	9,515,878	
364	45,624	.87	8,909,026	
395	49,426	.83	8,495,039	
424	53,228	.79	8,116,496	
456	57,030	.76	7,824,764	
486	60,832	.74	7,563,180	
517	64,634	.71	7,311,729	
547	68,436	.69	7,104,280	
577	72,238	.67	6,895,122	
608	76,040	.66	6,739,731	**200% of Current Size**

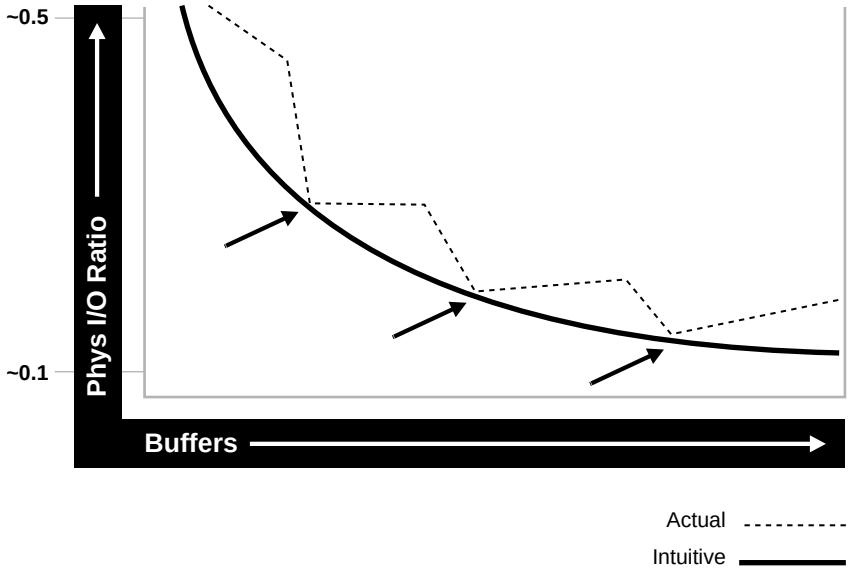
This view assists in cache sizing by providing information that predicts the number of *physical reads* for each potential cache size. The data also includes a *physical read factor*, which is a factor by which the current number of physical reads is estimated to change if the buffer cache is resized to a given value.

The relationship between successfully finding a block in the cache and the size of the cache is not always a smooth distribution. When sizing the buffer pool, avoid the use of additional buffers that contribute little or nothing to the cache hit ratio. In the example illustrated in Figure 14–1, only narrow bands of increments to the cache size may be worthy of consideration.

Examining Figure 14–1 leads to these observations:

- The benefit from increasing buffers from point A to point B is considerably higher than from point B to point C.
- The decrease in the physical I/O between points A and B and points B and C is not smooth, as indicated by the dotted line in the graph.

Figure 14–1 Physical I/O and Buffer Cache Size



Calculating the Buffer Cache Hit Ratio The buffer cache hit ratio calculates how often a requested block has been found in the buffer cache, without requiring disk access. This ratio is computed using data selected from the dynamic performance view V$SYSSTAT. The buffer cache hit ratio can be used to verify the physical I/O as predicted by V$DB_CACHE_ADVICE.

These statistics are used to calculate the hit ratio:

`session logical reads`	The total number of requests to access a block (whether in memory or on disk).
`physical reads`	The total number of requests to access a data block that resulted in access to datafiles on disk. (The block could have been read into the cache or read into local memory by a direct read).
`physical reads direct`	This indicates the number of blocks read, bypassing the buffer cache (excluding direct reads for LOBs).
`physical reads direct (lob)`	This indicates the number of blocks read while reading LOBs, bypassing the buffer cache.

The example below has been simplified by using values selected directly from the V$SYSSTAT table, without selecting these values over an interval. It is best to

calculate the delta of these statistics over an interval while your application is running, then use them in the formula below.

> **See Also:** Chapter 20, "Oracle Tools to Gather Database Statistics" for more information on collecting statistics over an interval

For example:

```
SELECT NAME, VALUE
  FROM V$SYSSTAT
WHERE NAME IN ('session logical reads','physical reads','physical reads
direct','physical reads direct (lob)')
```

The output of this query will look similar to the following:

```
NAME                                                                  VALUE
---------------------------------------------------------------- ----------
session logical reads                                             464905358
physical reads                                                     10380487
physical reads direct                                                 86850
physical reads direct (lob)                                               0
```

Calculate the hit ratio for the buffer cache with the following formula:

Hit Ratio = 1 - ((physical reads - physical reads direct - physical reads direct (lob)) / session logical reads)

Based on the sample statistics above, the buffer cache hit ratio is .978 (or 97.8%).

Interpreting and Using the Buffer Cache Statistics

There are a many factors to examine before considering whether or not to increase or decrease the buffer cache size. For example, examine V$DB_CACHE_ADVICE data and the buffer cache hit ratio before deciding to increase or decrease the buffer cache size.

Additionally, a low cache hit ratio does not imply that increasing the size of the cache would be beneficial for performance. A good cache hit ratio could wrongly indicate that the cache is adequately sized for the workload.

Considerations when looking at the buffer cache hit ratio include the following:

- Repeated scanning of the same large table or index could artificially inflate a poor cache hit ratio. Examine SQL statements with a large number of buffer gets that are executed frequently to ensure that the execution plan for such SQL statements is optimal. If possible, avoid repeatedly scanning

frequently-accessed data by performing all of the processing in a single pass, or by optimizing the SQL statement.

- If possible, avoid requerying the same data by caching frequently-accessed data in the client program or middle tier.
- Blocks encountered during a "long" full table scan are not put at the head of the LRU list. Therefore, the blocks are aged out faster than blocks read when performing indexed lookups or small table scans. Thus, poor hit ratios when valid large full table scans are occurring should also be considered when interpreting the buffer cache data.
- 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 might be little purpose in keeping the row or the block that contains it in memory for very long following its use.
- A common mistake is to continue increasing the buffer cache size. Such increases have no effect if you are doing full table scans or operations that do not use the buffer cache.

Increasing Memory Allocated to the Buffer Cache

As a general rule, investigate increasing the size of the cache if the cache hit ratio is low and your application has been tuned to avoid performing full table scans.

Set the DB_CACHE_ADVICE parameter to ON, and let the cache statistics stabilize. Examine the advisory data in the V$DB_CACHE_ADVICE view to determine the next increment required to significantly decrease the amount of physical I/O performed. If it is possible to allocate the required extra memory to the buffer cache, without causing the host OS to page, then allocate this memory. To increase the amount of memory allocated to the buffer cache, increase the value of the parameter DB_CACHE_SIZE.

If required, resize the buffer pools dynamically, rather than shutting down the instance to perform this change.

Note: When the cache is resized, DB_CACHE_ADVICE is set to OFF. Also, V$DB_CACHE_ADVICE shows the advisory for the old value of the cache. This remains until DB_CACHE_ADVICE is explicitly set back to READY or ON.

The DB_CACHE_SIZE parameter specifies the size of the default cache for the database's standard block size. To create and use tablespaces with block sizes different than the database's standard block sizes (such as to support transportable tablespaces), you must configure a separate cache for each block size used. The DB_*nK*_CACHE_SIZE parameter can be used to configure the size of the nonstandard block size needed (where *n* is 2, 4, 8, 16 or 32 and *n* is not the standard block size).

Note: The process of choosing a cache size is the same, regardless of whether the cache is the default standard block size cache, the KEEP or RECYCLE cache, or a nonstandard block size cache.

See Also: *Oracle9i Database Reference* and *Oracle9i Database Administrator's Guide* for more information on using the DB_*n*K_CACHE_SIZE parameters

Reducing Memory Allocated to the Buffer Cache

If your hit ratio is high, then your cache is probably large enough to hold your most frequently accessed data. Check V$DB_CACHE_ADVICE data to see whether decreasing your cache size significantly causes the number of physical I/Os to increase. If not, and if you require memory for another memory structure, then you might be able to reduce the cache size and still maintain good performance. To make the buffer cache smaller, reduce the size of the cache by changing the value for the parameter DB_CACHE_SIZE.

Considering Multiple Buffer Pools

A single default buffer pool is generally adequate for most systems. However, users with detailed knowledge of their application's buffer pool might benefit from configuring multiple buffer pools.

With segments that have atypical access patterns, cache blocks from those segments in two different buffer pools: the KEEP pool and the RECYCLE pool. A segment's access pattern may be atypical in that it is constantly accessed (that is, hot), or infrequently accessed (for example, a large segment accessed by a batch job only once per day).

Multiple buffer pools let you address these differences. You can use a KEEP buffer pool to maintain frequently-accessed segments in the buffer cache, and a RECYCLE buffer pool to prevent objects from consuming unnecessary space in the cache. When an object is associated with 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 a specific buffer pool. The default buffer pool is of size DB_CACHE_SIZE. Each buffer pool uses the same LRU replacement policy (for example, if the KEEP pool is not large enough to store all of the segments allocated to it, then the oldest blocks age out of the cache).

By allocating objects to appropriate buffer pools, you can do the following:

- Reduce or eliminate I/Os
- Isolate or limit an object to a separate cache

Random Access to Large Segments

A problem can occur with an LRU aging method when a very large segment is accessed with a large or unbounded index range scan. 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 it 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 issue exists for "warm" segments that are not accessed frequently enough to survive the buffer aging caused by the large segment reads. There are three options for solving this problem:

1. If the object accessed is an index, then investigate whether the index is selective. If not, then tune the SQL statement to use a more selective index.
2. If the SQL statement is tuned, then you can move the large segment into a separate RECYCLE cache, so that it does not affect 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.
3. Alternatively, 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 do not age out.

Oracle Real Application Cluster Instances

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 can be different sizes or not defined at all. Tune each instance according to the requirements placed by the application on that instance.

Buffer Pool data in V$DB_CACHE_ADVICE

V$DB_CACHE_ADVICE can be used to size all pools configured on your instance. Make the initial cache size estimate, run the representative workload, then simply query the V$DB_CACHE_ADVICE view for the pool you are interested in.

For example, to query data from the KEEP pool:

```
SELECT size_for_estimate, buffers_for_estimate
     , estd_physical_read_factor, estd_physical_reads
  FROM V$DB_CACHE_ADVICE
 WHERE name          = 'KEEP'
   AND block_size    = (SELECT value FROM V$PARAMETER
                          WHERE name = 'db_block_size')
   AND advice_status = 'ON';
```

Buffer Pool Hit Ratios

The data in V$SYSSTAT reflects the logical and physical reads for all buffer pools within one set of statistics. To determine the hit ratio for the buffer pools individually, you must query the V$BUFFER_POOL_STATISTICS view. This view maintains per-pool statistics on the number of logical reads and writes.

The buffer pool hit ratio can be determined using the following formula:

hit ratio = 1 - [physical reads/(block gets + consistent gets)]

where the values of physical reads, block gets, and consistent gets can be obtained from the following query:

```
SELECT NAME, PHYSICAL_READS, DB_BLOCK_GETS, CONSISTENT_GETS
     , 1 - (PHYSICAL_READS / (DB_BLOCK_GETS + CONSISTENT_GETS)) "Hit Ratio"
  FROM V$BUFFER_POOL_STATISTICS;
```

Determining Which Segments Have Many Buffers in the Pool

The V$BH view shows the data object ID of all blocks that currently reside in the SGA. You can either look at the buffer cache usage pattern of all segments using

Method 1, or examine the usage pattern of a specific segment, as shown in Method 2.

Method 1 The query below counts the number of blocks for *all segments* that reside in the buffer cache at that point in time. Depending on your buffer cache size, this could require a lot of sort space:

```
column object_name format a40
column number_of_blocks format 999,999,999,999

SELECT o.object_name, COUNT(1) number_of_blocks
  FROM DBA_OBJECTS o, V$BH bh
 WHERE o.object_id  = bh.objd
   AND o.owner     != 'SYS'
 GROUP BY o.object_name
 ORDER BY count(1);

OBJECT_NAME                              NUMBER_OF_BLOCKS
---------------------------------------- ----------------
OA_PREF_UNIQ_KEY                                       1
SYS_C002651                                            1
..
DS_PERSON                                             78
OM_EXT_HEADER                                        701
OM_SHELL                                           1,765
OM_HEADER                                          5,826
OM_INSTANCE                                       12,644
```

Method 2 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 = upper('<SEGMENT_NAME>');
   ```

   Because two objects can have the same name (if they are different types of objects), 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 number of buffers in the instance:

```
SELECT name, block_size, sum(buffers)
  FROM V$BUFFER_POOL
 GROUP BY name, block_size
 HAVING SUM(buffers) > 0;
```

4. Calculate the ratio of buffers to total buffers to obtain the percentage of the cache currently used by SEGMENT_NAME.

   % cache used by segment_name = [buffers(Step2)/total buffers(Step3)]

> **Note:** This technique works only for a single segment. You must run the query for each partition for a partitioned object.

Keep Pool

If there are certain segments in your application that are referenced frequently, then cache blocks from those segments in a separate cache called the KEEP buffer pool. Memory is allocated to the KEEP buffer pool by setting the parameter DB_KEEP_POOL_SIZE to the required size. The memory for the KEEP pool is not a subset of the default pool. Typical segments that can be kept are small reference tables, which are used frequently. Application developers and DBAs can determine which tables are candidates.

You can check the number of blocks from candidate tables by querying V$BH, as described in "Determining Which Segments Have Many Buffers in the Pool" on page 14-13.

> **Note:** The NOCACHE clause has no effect on a table in the KEEP cache.

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 blocks used by all objects assigned to this

pool. If you gather statistics on the segments, you can query DBA_TABLES.BLOCKS and DBA_TABLES.EMPTY_BLOCKS to determine the number of blocks used.

Calculate the hit ratio by taking two snapshots of system performance at different times using the previous 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 might 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 might 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 Pool

It is possible to configure a RECYCLE buffer pool for blocks belonging to those segments that you do not want to remain in memory. The RECYCLE pool is good for segments that are scanned rarely or are not referenced frequently. 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 need not be cached; those cache buffers can be allocated to other objects.

Memory is allocated to the RECYCLE buffer pool by setting the parameter DB_RECYCLE_POOL_SIZE to the required size. This memory for the RECYCLE buffer pool is not a subset of the default pool.

Do not discard blocks from memory too quickly. If the buffer pool is too small, then blocks can age out of the cache before the transaction or SQL statement has completed execution. For example, an application might 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.

Using Multiple Buffer Pools

To define a default buffer pool for an object, use the BUFFER_POOL keyword of the STORAGE clause. This clause is valid for CREATE and ALTER TABLE, CLUSTER, and INDEX SQL statements. After a BUFFER_POOL has been specified, all subsequent blocks read for the object are placed in that 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 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.

> **See Also:** *Oracle9i SQL Reference* for information on specifying BUFFER_POOL in the STORAGE clause

Configuring and Using the Shared Pool and Large Pool

Oracle uses the shared pool to cache many different types of data. Cached data includes the textual and executable forms of PL/SQL blocks and SQL statements, dictionary cache data, and other data.

Proper use and sizing of the shared pool can reduce resource consumption in at least four ways:

1. Parse overhead is avoided if the SQL statement is already in the shared pool. This saves CPU resources on the host and elapsed time for the end user.
2. Latching resource usage is significantly reduced, which results in greater scalability.
3. Shared pool memory requirements are reduced, because all applications use the same pool of shared SQL statements and dictionary resources.
4. I/O resources are saved, because dictionary elements that are in the shared pool do not require disk access.

This section covers the following:

- Shared Pool Concepts
- Using the Shared Pool Effectively
- Sizing the Shared Pool
- Interpreting Shared Pool Statistics
- Consider using the Large Pool
- Consider Using CURSOR_SPACE_FOR_TIME
- Consider Caching Session Cursors
- Consider Configuring the Reserved Pool
- Consider Keeping Large Objects to Prevent Aging
- Consider CURSOR_SHARING for Existing Applications

Shared Pool Concepts

The main components of the shared pool are the library cache and the dictionary cache. The library cache stores the executable (parsed or compiled) form of recently referenced SQL and PL/SQL code. The dictionary cache stores data referenced from the data dictionary. Many of the caches in the shared pool automatically increase or decrease in size, as needed, including the library cache and the dictionary cache. Old entries are aged out of these caches to accommodate new entries when the shared pool does not have free space for a new entry.

A cache miss on the data dictionary cache or library cache is more expensive than a miss on the buffer cache. For this reason, the shared pool should be sized to ensure that frequently used data is cached.

There are a number of features that make large memory allocations in the shared pool, such as the shared server, parallel query, or Recovery Manager. Oracle recommends segregating the SGA memory used by these features by configuring a distinct memory area, called the *large pool*.

> **See Also:** "Consider using the Large Pool" on page 14-33 for more information on configuring the large pool

Allocation of memory from the shared pool is performed in chunks. This allows large objects (over 5k) to be loaded into the cache without requiring a single

contiguous area, hence reducing the possibility of running out of enough contiguous memory due to fragmentation.

Infrequently, Java, PL/SQL, or SQL cursors may make allocations out of the shared pool that are larger than 5k. To allow these allocations to happen most efficiently, Oracle segregates a small amount of the shared pool. This memory is used if the shared pool does not have enough space. The segregated area of the shared pool is called the *reserved pool*.

> **See Also:** "Consider Configuring the Reserved Pool" on page 14-38 for more information on the reserved area of the shared pool

Dictionary Cache Concepts

Information stored in the data dictionary cache includes usernames, segment information, profile data, tablespace information, and sequence numbers. The dictionary cache 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.

Library Cache Concepts

The library cache holds executable forms of SQL cursors, PL/SQL programs, and Java classes. This section focuses on tuning as it relates to cursors, PL/SQL programs, and Java classes. These are collectively referred to as *application code*.

When application code is run, Oracle attempts to reuse existing code if it has been executed previously and can be shared. If the parsed representation of the statement does exist in the library cache and it can be shared, then Oracle reuses the existing executable. This is known as a *soft parse*, or a *library cache hit*.

If Oracle is unable to use existing code, then a new executable version of the application code must be built. This is known as a *hard parse*, or a *library cache miss*.

> **See Also:** "SQL Sharing Criteria" on page 14-20 for details on when a SQL and PL/SQL statements can be shared

Library cache misses can occur on either the parse step or the execute step when processing a SQL statement.

When an application makes a *parse* call for a SQL statement, if the parsed representation of the statement does not already exist in the library cache, then Oracle parses the statement and stores the parsed form in the shared pool. This is a

hard parse. You might be able to reduce library cache misses on parse calls by ensuring that all shareable SQL statements are in the shared pool whenever possible.

If an application makes an *execute* call for a SQL statement, and if the executable portion of the previously built SQL statement has been aged out (that is, deallocated) from the library cache to make room for another statement, then Oracle implicitly reparses the statement, creating a new shared SQL area for it, and executes it. This also results in a hard parse. Usually, you can reduce library cache misses on execution calls by allocating more memory to the library cache.

In order to perform a hard parse, Oracle uses more resources than during a soft parse. Resources used for a soft parse include CPU and library cache latch gets. Resources required for a hard parse include additional CPU, library cache latch gets, and shared pool latch gets.

SQL Sharing Criteria

Oracle automatically determines whether a SQL statement or PL/SQL block being issued is identical to another statement currently in the shared pool.

Oracle's performs the following steps for the comparison:

1. The text of the statement issued is compared to existing statements in the shared pool.

   The text of the statement is hashed. If there is no matching hash value, then the SQL statement does not currently exist in the shared pool, and a hard parse is performed.

   If there is a matching hash value for an existing SQL statement in the shared pool, then Oracle compares the text of the matched statement to the text of the statement hashed to see if they are identical. The text of the SQL statements or PL/SQL blocks must be identical, character for character, including spaces, case, and comments. For example, the following statements *cannot* use the same shared SQL area:

   ```
   SELECT * FROM employees;
   SELECT * FROM Employees;
   SELECT *  FROM employees;
   ```

   Usually, SQL statements that differ only in literals cannot use the same shared SQL area. For example, the following SQL statements do not resolve to the same SQL area:

```
SELECT count(1) FROM employees where manager_id = 121;
SELECT count(1) FROM employees where manager_id = 247;
```

The only exception to this is when the parameter CURSOR_SHARING has been set to SIMILAR or FORCE. *Similar* statements also share SQL areas when the CURSOR_SHARING parameter is set to SIMILAR or FORCE. The costs and benefits involved in using CURSOR_SHARING are explained later in this section.

> **See Also:** *Oracle9i Database Reference* for more information on the CURSOR_SHARING parameter

2. References to schema objects in the SQL statements or PL/SQL blocks must resolve to the same object in the same schema.

   The objects referenced in the issued statement are compared to the referenced objects of all existing statements identified above to ensure that they are identical.

   For example, if two users each issue the following SQL statement:

   ```
   SELECT * FROM employees;
   ```

   and they each have *their own* employees table, then the above statement is not considered identical, because the statement references different tables for each user.

3. Bind variables in the SQL statements must match in name, datatype, and length.

   For example, the following statements cannot use the same shared SQL area, because the bind variable names differ:

   ```
   SELECT * FROM employees WHERE department_id = :department_id;
   SELECT * FROM employees WHERE department_id = :dept_id;
   ```

   Many Oracle products (such as Oracle Forms and the precompilers) convert the SQL before passing statements to the database. Characters are uniformly changed to uppercase, white space is compressed, and bind variables are renamed so that a consistent set of SQL statements is produced.

4. The session's environment must be identical. Items compared include the following:

- Optimization approach and goal. SQL statements must be optimized using the same optimization approach and, in the case of the cost-based approach, the same optimization goal.
- Session-configurable parameters such as `SORT_AREA_SIZE`.

Using the Shared Pool Effectively

An important purpose of the shared pool is to cache the executable versions of SQL and PL/SQL statements. This allows multiple executions of the same SQL or PL/SQL code to be performed *without* the resources required to hard parse, which results in significant reductions in CPU, memory, and latch usage.

The shared pool is also able to support unshared SQL in data warehousing applications, which execute low-concurrency high-resource SQL statements. In this situation, using unshared SQL with literal values is recommended. Using literal values rather than bind variables allows the optimizer to make good column selectivity estimates, thus providing an optimal data access plan.

> **See Also:** *Oracle9i Data Warehousing Guide*

In an OLTP system, there are a number of ways to ensure efficient use of the shared pool and related resources. Discuss the following possibilities with application developers and agree on strategies to ensure that the shared pool is used effectively:

- Share Cursors
- Maintain Connections
- Single-user Log on and Qualified Table Reference
- Use of PL/SQL
- Avoid Performing DDL
- Cache Sequence Numbers
- Cursor Access and Management

Efficient use of the shared pool in high-concurrency OLTP systems significantly reduces the probability of parse-related application scalability issues.

Share Cursors

Reuse of shared SQL for multiple users running the same application, avoids hard parsing. Soft parses provide a significant reduction in resource usage, such as shared pool and library cache latches. To share cursors, do the following:

- Use bind variables rather than literals in SQL statements whenever possible. For example, the following two statements cannot use the same shared area because they do not match character for character:

```
SELECT employee_id FROM employees WHERE department_id = 10;
SELECT employee_id FROM employees WHERE department_id = 20;
```

  By replacing the literals with a bind variable, only one SQL statement would result, which could be executed twice:

```
SELECT employee_id FROM employees WHERE department_id = :dept_id;
```

  Note: For existing applications where rewriting the code to use bind variables is impractical, it is possible to use the CURSOR_SHARING initialization parameter to avoid some of the hard parse overhead. For more information see section "Consider CURSOR_SHARING for Existing Applications" on page 14-42.

- Avoid application designs that result in large numbers of users issuing dynamic, unshared SQL statements. Typically, the majority of data required by most users can be satisfied using preset queries. Use dynamic SQL where such functionality is required.
- Be sure that users of the application do not change the optimization approach and goal for their individual sessions.
- Establish the following policies among the developers of the applications:
  - Standardize naming conventions for bind variables and spacing conventions for SQL statements and PL/SQL blocks.
  - Consider using 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, their use reduces run-time parsing.

Maintain Connections

Large OLTP applications with middle-tiers should maintain connections, rather than connecting and disconnecting for each database request. Maintaining connections saves CPU resources and database resources, such as latches.

Single-user Log on and Qualified Table Reference

Large OLTP systems where users log in to the database as their own userID can benefit from qualifying the segment owner explicitly, rather than by using public synonyms. This significantly reduces the number of entries in the dictionary cache. For example:

```
SELECT employee_id FROM hr.employees WHERE department_id = :dept_id;
```

An alternative to qualifying table names is to connect to the database via a single userID, rather than individual userIDs. User-level validation can take place locally on the middle-tier. Reducing the number of distinct userIDs also reduces the load on the dictionary cache.

Use of PL/SQL

Using stored PL/SQL packages can overcome much of the scalability issues for systems with thousands of users, each with individual user sign-on and public synonyms. This is because a package is executed as the owner, rather than the caller, which reduces dictionary cache load considerably.

Avoid Performing DDL

Avoid performing DDL operations on high-usage segments during peak hours. Performing DDL on such segments often results in the dependent SQL being invalidated, and hence reparsed on a later execution.

Cache Sequence Numbers

Allocating sufficient cache for frequently updated sequence numbers significantly reduces the frequency of dictionary cache locks, which improves scalability. The CACHE keyword on the CREATE SEQUENCE or ALTER SEQUENCE statement lets you configure the number of cached entries for each sequence.

> **See Also:** *Oracle9i SQL Reference* for details on the CREATE SEQUENCE and ALTER SEQUENCE statements

Cursor Access and Management

Depending on the Oracle application tool you are using, it is possible to control how frequently your application performs parse calls.

The frequency with which your application either closes cursors or reuses existing cursors for new SQL statements affects the amount of memory used by a session, and often the amount of parsing performed by that session.

An application that closes cursors or reuses cursors (for a different SQL statement, rather than the same SQL statement), does not need as much session memory as an application that keeps cursors open. Conversely, that same application may need to perform more parse calls, hence using extra CPU and Oracle resources.

Cursors associated with SQL statements that are *not* executed frequently can be closed or reused for other statements, because the likelihood of reexecuting (and reparsing) that statement is low.

Extra parse calls are required when a cursor containing a SQL statement that *will be reexecuted* is closed or reused for another statement. Had the cursor remained open, it could have been reused without the overhead of issuing a parse call.

The ways in which you control cursor management depends on your application development tool. The following section introduces the methods used for some Oracle tools.

See Also:

- The tool-specific documentation for more information about each tool
- *Oracle9i Database Performance Methods* for more information on cursor sharing and management

Reducing Parse Calls with OCI Do not close and reopen cursors that you will be reexecuting. Instead, leave the cursors open, and change the literal values in the bind variables before execution.

Avoid reusing statement handles for new SQL statements when the existing SQL statement will be reexecuted in the future.

Reducing Parse Calls with the Oracle Precompilers When using the Oracle precompilers, you can control when cursors are closed by setting precompiler clauses. In Oracle mode, the clauses 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 on the precompiler command line or within the precompiler program. With these clauses, you can employ different strategies for managing cursors during execution of the program.

> **See Also:** Your language's precompiler manual more information on these clauses

Reducing Parse Calls with SQLJ Prepare the statement, then reexecute the statement with the new values for the bind variables. The cursor stays open for the duration of the session.

Reducing Parse Calls with JDBC Avoid closing cursors if they will be reexecuted, because the new literal values can be bound to the cursor for reexecution. Alternatively, JDBC provides a SQL Statement Cache within the JDBC client using the `setStmtCacheSize()` method. Using this method, JDBC creates a SQL statement cache that is local to the JDBC program.

> **See Also:** *Oracle9i JDBC Developer's Guide and Reference* manual for more information on using the JDBC SQL statement cache

Reducing Parse Calls with Oracle Forms With Oracle Forms, it is possible to control some aspects of cursor management. You can exercise this control either at the trigger level, at the form level, or at run time.

> **See Also:** *Oracle Forms Reference* manual for more information on the reuse of private SQL areas by Oracle Forms

Sizing the Shared Pool

When configuring a brand new instance, it is impossible to know the correct size to make the shared pool cache. Typically, a DBA makes a first estimate for the cache size, then runs a representative workload on the instance, and examines the relevant statistics to see whether the cache is under- or over-configured.

For most OLTP applications, shared pool size is an important factor to the application performance. Shared pool size is less important for applications that issue a very limited number of discrete SQL statements, such as DSS systems.

If the shared pool is too small, then extra resources are used to manage the limited amount of available space. This consumes CPU and latching resources, and causes contention.

Optimally, the shared pool should be just large enough to cache frequently accessed objects. Having a significant amount of free memory in the shared pool is a waste of memory.

Shared Pool - Library Cache Statistics

The goal when sizing the library cache is to ensure that SQL statements that will be executed multiple times are cached, without allocating too much memory.

The statistic that shows the amount of reloading (that is, reparsing) of a previously cached SQL statement that was aged out of the cache is the RELOADS column in the V$LIBRARYCACHE view. In an application that reuses SQL effectively, on a system with an optimal shared pool size, the RELOADS statistic would have a value of zero.

The INVALIDATIONS column in this view shows the number of times library cache data was invalidated and had to be reparsed. INVALIDATIONS should be near zero. This means SQL statements that could have been shared were invalidated by some operation (for example, a DDL). This statistic should be near zero on OLTP systems during peak loads.

Another key statistic is the amount of `free memory` in the shared pool at peak times. The amount of free memory can be queried from V$SGASTAT, looking at the `free memory` for the `shared pool`. Optimally, free memory should be as low as possible, without causing any RELOADS on the system.

Lastly, a broad indicator of library cache health is the `library cache hit ratio`. This value should be considered along with the above statistics and other data, such as the rate of hard parsing and whether there is any shared pool or library cache latch contention.

These statistics are discussed below in more detail.

V$LIBRARYCACHE

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.

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 view 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.

> **See Also:** Chapter 24, "Dynamic Performance Views for Tuning" for information on columns of the V$LIBRARYCACHE view

To examine each namespace individually, use the following query:

```
SELECT namespace
     , pins
     , pinhits
     , reloads
     , invalidations
  FROM V$LIBRARYCACHE
 ORDER BY namespace;
```

The output of this query could look like the following:

```
NAMESPACE             PINS     PINHITS     RELOADS INVALIDATIONS
--------------- ---------- ---------- ---------- -------------
BODY                  8870       8819          0             0
CLUSTER                393        380          0             0
INDEX                   29          0          0             0
OBJECT                   0          0          0             0
PIPE                 55265      55263          0             0
SQL AREA          21536413   21520516      11204             2
TABLE/PROCEDURE   10775684   10774401          0             0
TRIGGER               1852       1844          0             0
```

To calculate the library cache hit ratio, use the following formula:

Library Cache Hit Ratio = sum(pinhits) / sum(pins)

Using the library cache hit ratio formula above, the cache hit ratio is the following:

```
SUM(PINHITS)/SUM(PINS)
----------------------
            .999466248
```

Note: The queries return data from instance startup, rather than statistics gathered during an interval, which is preferable.

See Also: Chapter 20, "Oracle Tools to Gather Database Statistics" for information on how gather information over an interval

Examining the data returned leads to these observations:

- For the SQL Area namespace, there were 21,536,413 EXECUTIONS.
- 11,204 of the executions resulted in a library cache miss, requiring Oracle to implicitly reparse a statement or block or reload an object definition because it aged out of the library cache (that is, a RELOAD).
- SQL statement(s) were invalidated two times, again causing library cache misses.
- The hit percentage is about 99.94%. This means that only .06% of executions resulted in reparsing.

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'
   AND POOL = 'shared pool';
```

The output will be similar to the following:

```
POOL        NAME                            BYTES
----------- -------------------------- ----------
shared pool free memory                   4928280
```

If there is always free memory available in 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. It may be indicative of a well-configured system.

Shared Pool - Dictionary Cache Statistics

The algorithm Oracle uses to manage data in the shared pool ages out library cache data in preference to aging out dictionary cache data. Therefore, configuring the

library cache to an acceptable size usually ensures that the data dictionary cache is also adequately sized.

Misses on the data dictionary cache are to be expected in some cases. On instance startup, the data dictionary cache contains no data. Therefore, any SQL statement issued is likely to result in cache misses. As more data is read into the cache, the likelihood of cache misses decreases. Eventually, the database reaches a *steady state*, in which the most frequently used dictionary data is in the cache. At this point, very few cache misses occur.

Each row in the V$ROWCACHE view contains statistics for a single type of the data dictionary item. These statistics reflect all data dictionary activity since the most recent instance startup. These columns in the V$ROWCACHE view 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 which were not satisfied by the cache (requiring an I/O).
MODIFICATIONS	Shows the number of times data in the dictionary cache was updated.

Use the following query to monitor the statistics in the V$ROWCACHE. view over a period of time while your application is running. The derived column PCT_SUCC_GETS can be considered the item-specific hit ratio:

```
column parameter format a21
column pct_succ_gets format 999.9
column updates format 999,999,999

SELECT parameter
     , sum(gets)
     , sum(getmisses)
     , 100*sum(gets - getmisses) / sum(gets)  pct_succ_gets
     , sum(modifications)                     updates
  FROM V$ROWCACHE
 WHERE gets > 0
 GROUP BY parameter;
```

The output of this select will be similar to the following:

```
PARAMETER             SUM(GETS) SUM(GETMISSES) PCT_SUCC_GETS      UPDATES
-------------------- ---------- -------------- ------------ ------------
dc_database_links            81              1         98.8            0
dc_free_extents           44876          20301         54.8       40,453
dc_global_oids               42              9         78.6            0
dc_histogram_defs          9419            651         93.1            0
dc_object_ids             29854            239         99.2           52
dc_objects                33600            590         98.2           53
dc_profiles               19001              1        100.0            0
dc_rollback_segments      47244             16        100.0           19
dc_segments              100467          19042         81.0       40,272
dc_sequence_grants          119             16         86.6            0
dc_sequences              26973             16         99.9       26,811
dc_synonyms                6617            168         97.5            0
dc_tablespace_quotas        120              7         94.2           51
dc_tablespaces           581248             10        100.0            0
dc_used_extents           51418          20249         60.6       42,811
dc_user_grants            76082             18        100.0            0
dc_usernames             216860             12        100.0            0
dc_users                 376895             22        100.0            0
```

Examining the data returned by the sample query leads to these observations:

- There are large numbers of misses and updates for used extents, free extents, and segments. This implies that the instance had a significant amount of dynamic space extension.
- Based on the percentage of successful gets and comparing that statistic with the actual number of gets, the shared pool is large enough to store dictionary cache data adequately.

It is also possible to calculate an overall dictionary cache hit ratio as shown below, although summing up the data over all the caches will lose the finer granularity of data:

```
SELECT (SUM(GETS - GETMISSES - FIXED)) / SUM(GETS) "ROW CACHE" FROM V$ROWCACHE;
```

Interpreting Shared Pool Statistics

Increasing Memory Allocation

Increasing the amount of memory for the shared pool increases the amount of memory usable by both the library cache and the dictionary cache.

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 zero. 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 might 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.

Allocating Additional Memory to the Data Dictionary Cache Examine cache activity by monitoring 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%, depending on the application.

Consider increasing the amount of memory available to the cache if all of the following are true:

- Your application is using the shared pool effectively (see "Using the Shared Pool Effectively" on page 14-22).
- Your system has reached a steady state, any of the item-specific hit ratios are low, and there are a large numbers of gets for the caches with low hit ratios.

Increase the amount of memory available to the data dictionary cache by increasing the value of the initialization parameter SHARED_POOL_SIZE.

Reducing Memory Allocation

If your RELOADS are near zero, and if you have a small amount of free memory in the shared pool, then the shared pool is probably large enough to hold the most frequently accessed data.

If you always have significant amounts of memory free in the shared pool, and if you would like to allocate this memory elsewhere, then you might be able to reduce the shared pool size and still maintain good performance.

To make the shared pool smaller, reduce the size of the cache by changing the value for the parameter SHARED_POOL_SIZE.

Consider using the Large Pool

Unlike the shared pool, the large pool does not have an LRU list. Oracle does not attempt to age memory out of the large pool.

You should consider configuring a large pool if your instance uses any of the following:

- Parallel Query

  Parallel query uses shared pool memory to cache parallel execution message buffers.

  > **See Also:** *Oracle9i Data Warehousing Guide* for more information on sizing the large pool with parallel query

- Recovery Manager

  Recovery Manager uses the shared pool to cache I/O buffers during backup and restore operations. For I/O server processes and backup and restore operations, Oracle allocates buffers that are a few hundred kilobytes in size.

  > **See Also:** *Oracle9i Recovery Manager User's Guide and Reference* for more information on sizing the large pool when using Recovery Manager

- Shared Server

  In a shared server architecture, the session memory for each client process is included in the shared pool.

Tuning the Large Pool and Shared Pool for the Shared Server Architecture

As Oracle allocates shared pool memory for shared server session memory, the amount of shared pool memory available for the library cache and dictionary cache decreases. If you allocate this session memory from a different pool, then Oracle can use the shared pool primarily for caching shared SQL and not incur the performance overhead from shrinking the shared SQL cache.

Oracle recommends using the large pool to allocate shared server-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 decreases fragmentation of the shared pool.

To store shared server-related UGA in the large pool, specify a value for the initialization parameter LARGE_POOL_SIZE. To see which pool (shared pool or large pool) the memory for an object resides, check the column POOL in V$SGASTAT. The large pool is not configured by default; its minimum value is 300K. If you do not configure the large pool, then Oracle uses the shared pool for shared server user session memory.

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, assuming that the shared server requires 200K - 300K to store session information for each active session. If you anticipate 100 active sessions simultaneously, then configure the large pool to be 30M, or increase the shared pool accordingly if the large pool is not configured.

Note: If a shared server architecture 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 CIRCUITS initialization parameter specifies the maximum number of concurrent shared server connections that the database allows.

See Also:

- *Oracle9i Database Concepts* for more information about the large pool
- *Oracle9i Database Reference* for complete information about initialization parameters

Determining an Effective Setting for Shared Server 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 shared servers, the total amount of memory use decreases. This is because there are fewer processes, and therefore Oracle uses less PGA memory with shared servers when compared to dedicated server environments.

Note: For best performance with sorts using shared servers, 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.

Checking System Statistics in the V$SESSTAT View Oracle collects statistics on total memory used by a session and stores them in the dynamic performance view V$SESSTAT:

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.

You can use the following query to decide how much larger to make the shared pool if you are using a shared server. Issue the following 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 view V$STATNAME to obtain internal identifiers for *session memory* and *max session memory*. The results of these queries could look like the following:

```
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 shared server architecture. 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.

Limiting Memory Use Per User Session by Setting PRIVATE_SGA You can set the PRIVATE_SGA resource limit to restrict 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 used rarely, because most DBAs do not limit SGA consumption an a user-by-user basis.

> **See Also:** *Oracle9i SQL Reference*, ALTER RESOURCE COST statement, for more information about setting the PRIVATE_SGA resource limit

Reducing Memory Use With Three-Tier Connections If you have a high number of connected users, then you can reduce memory usage 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. A shared server environment 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. A shared server environment also substantially reduces overall memory usage even though more SGA is used in shared server mode.

Consider Using CURSOR_SPACE_FOR_TIME

If you have no library cache misses, then you might be able to accelerate execution calls by setting the value of the initialization parameter `CURSOR_SPACE_FOR_TIME` to `true`. This parameter specifies whether a cursor 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 cursor 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 the cursor containing the SQL statement is in the library cache.
- If this is set to `true`, then a cursor can be deallocated only when all application cursors associated with its statement are closed. In this case, Oracle need not verify that a cursor is in the cache, because it cannot 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 can slightly improve the performance of execution calls. This value also prevents the deallocation of cursors 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 cursor. Although deallocating a cursor could result in a library cache miss later (only if the cursor is reexecuted), 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 your available memory so

that there is no space for a new SQL statement, then the statement cannot be parsed. Oracle returns an error indicating that there is not enough memory.

Consider 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 checks the library cache to determine whether more than three parse requests have been issued on a given statement. If so, then Oracle assumes that 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 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 = <value>;
```

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$SYSSTAT 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 consider setting SESSION_CACHED_CURSORS to a larger value.

Consider Configuring the Reserved Pool

Although Oracle breaks down very large requests for memory into smaller chunks, on some systems there might still be a requirement to find a contiguous chunk (for example, over 5k) of memory. (The default minimum reserved pool allocation is 4400.)

If there is not enough free space in the shared pool, then Oracle must search for and free enough memory to satisfy this request. This operation could conceivably hold

latch resource for detectable periods of time, causing minor disruption to other concurrent attempts at memory allocation.

The most efficient way of allocating large chunks is to have a small reserved memory area in the shared pool that can be used if the shared pool does not have enough space. This is called the *reserved pool*.

The reserved pool area of the shared pool provides an additional memory area that is used if there is not enough space in the shared pool. This memory area avoids the overheads required to find and create space for large allocations in the shared pool. Small objects cannot fragment the reserved list, because they cannot use the reserved pool, helping to ensure that the reserved pool has large contiguous chunks of memory.

By default, Oracle configures a small reserved pool. This memory can be used for operations such as temporary space while loading Java objects or PL/SQL and trigger compilation. After the memory allocated from the reserved pool is freed, it returns to the reserved pool.

It is unlikely that you need to change the default amount of space Oracle reserves. However, if necessary, the reserved pool size can be changed by setting the SHARED_POOL_RESERVED_SIZE initialization parameter. This parameter sets aside space in the shared pool for unusually large allocations.

For large allocations, the order in which Oracle attempts to allocate space in the shared pool is the following:

1. From the unreserved part of the shared pool.
2. If there is not enough space in the unreserved part of the shared pool, and if the allocation is large, then Oracle checks whether the reserved pool has enough space.
3. If there is not enough space in the unreserved and reserved parts of the shared pool, then Oracle attempts to free enough memory for the allocation. It then retries the unreserved and reserved parts of the shared pool.

Using SHARED_POOL_RESERVED_SIZE The default value for SHARED_POOL_RESERVED_SIZE is 5% of the SHARED_POOL_SIZE. This means that, by default, the reserved list is configured.

If you set SHARED_POOL_RESERVED_SIZE > 1/2 SHARED_POOL_SIZE, then Oracle signals an error. Oracle does not let you reserve too much memory for the reserved pool. The amount of operating system memory, however, might 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 takes memory from the shared pool. (This reduces the amount of unreserved shared pool memory available for smaller allocations.)

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:** *Oracle9i Database Reference* for details on setting the LARGE_POOL_SIZE parameter

When SHARED_POOL_RESERVED_SIZE is 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.

Increasing the value of SHARED_POOL_RESERVED_SIZE 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.

When SHARED_POOL_RESERVED_SIZE is Too Large Too much memory might 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.

When SHARED_POOL_SIZE is Too Small The V$SHARED_POOL_RESERVED fixed view can also indicate when the value for SHARED_POOL_SIZE is too small. This can 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.

Consider Keeping Large Objects to Prevent Aging

After an entry has been loaded into the shared pool, it cannot be moved. Sometimes, as entries are loaded and aged, the free memory can become fragmented.

Use the PL/SQL package DBMS_SHARED_POOL to manage the shared pool. 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 might want to prevent large SQL or PL/SQL areas from aging out of the shared pool.

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.

The DBMS_SHARED_POOL package is useful for the following:

- When loading large PL/SQL objects, such as the STANDARD and DIUTIL packages. When large PL/SQL objects are loaded, user response time may be affected if smaller objects that need to age out of the shared pool to make room. In some cases, there might be insufficient memory to load the large objects.
- Frequently executed triggers. You might want to keep compiled triggers on frequently used tables in the shared pool.
- DBMS_SHARED_POOL 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.

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 kept 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:** *Oracle9i Supplied PL/SQL Packages Reference* for specific information on using DBMS_SHARED_POOL procedures

Consider CURSOR_SHARING for Existing Applications

One of the first stages of parsing is to compare the text of the statement with existing statements in the shared pool to see if the statement can be shared. If the statement differs textually in any way, then Oracle does not share the statement.

Exceptions to this are possible when the parameter CURSOR_SHARING has been set to SIMILAR or FORCE. When this parameter is used, Oracle first checks the shared pool to see if there is an identical statement in the shared pool. If an identical statement is not found, then Oracle searches for a similar statement in the shared pool. If the similar statement is there, then the parse checks continue to verify the executable form of the cursor can be used. If the statement is not there, then a hard parse is necessary to generate the executable form of the statement.

Similar SQL Statements

Statements that are identical, except for the values of some literals, are called *similar* statements. Similar statements pass the textual check in the parse phase when the CURSOR_SHARING parameter is set to SIMILAR or FORCE. Textual similarity does not guarantee sharing. The new form of the SQL statement still needs to go through the remaining steps of the parse phase to ensure that the execution plan of the preexisting statement is equally applicable to the new statement.

> **See Also:** "SQL Sharing Criteria" on page 14-20 for more details on the various checks performed

CURSOR_SHARING

Setting CURSOR_SHARING to EXACT allows SQL statements to share the SQL area only when their texts match exactly. This is the default behavior. Using this setting, similar statements cannot shared; only textually exact statements can be shared.

Setting CURSOR_SHARING to either SIMILAR or FORCE allows similar statements to share SQL. The difference between SIMILAR and FORCE is that SIMILAR forces similar statements to share the SQL area without deteriorating execution plans. Setting CURSOR_SHARING to FORCE forces similar statements to share the executable SQL area, potentially deteriorating execution plans. Hence, FORCE should be used as a last resort, when the risk of suboptimal plans is outweighed by the improvements in cursor sharing.

When to use CURSOR_SHARING

The CURSOR_SHARING initialization parameter can solve some performance problems. It has the following values: FORCE, SIMILAR, and EXACT (default). Using this parameter provides benefit to existing applications that have many similar SQL statements.

Note: Oracle does not recommend setting CURSOR_SHARING to FORCE in a DSS environment or if you are using complex queries. Also, star transformation is not supported with CURSOR_SHARING set to either SIMILAR or FORCE. For more information, see the "OPTIMIZER_FEATURES_ENABLE Parameter" on page 1-66.

The optimal solution is to write sharable SQL, rather than rely on the CURSOR_SHARING parameter. This is because although CURSOR_SHARING does significantly reduce the amount of resources used by eliminating hard parses, it requires some extra work as a part of the soft parse to find a similar statement in the shared pool.

Note: Setting CURSOR_SHARING to SIMILAR or 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 does not change.

Consider setting CURSOR_SHARING to SIMILAR or 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?

Caution: Setting CURSOR_SHARING to FORCE or SIMILAR prevents any outlines generated with literals from being used if they were generated with CURSOR_SHARING set to EXACT.

To use stored outlines with CURSOR_SHARING=FORCE or SIMILAR, the outlines must be *generated* with CURSOR_SHARING set to FORCE or SIMILAR and with the CREATE_STORED_OUTLINES parameter.

Using CURSOR_SHARING = SIMILAR (or FORCE) can significantly improve cursor sharing on some applications that have many similar statements, resulting in reduced memory usage, faster parses, and reduced latch contention.

Configuring and Using the Java Pool

If your application uses Java, you should investigate whether you need to modify the default configuration for the Java pool.

See Also: *Oracle9i Java Developer's Guide*

Configuring and Using the Redo Log Buffer

Server processes making changes to data blocks in the buffer cache generate redo data into the log buffer. LGWR begins writing to copy entries from the redo log buffer to the online redo log if any of the following are true:

- The log buffer becomes one third full.
- LGWR is posted by a server process performing a COMMIT or ROLLBACK.
- DBWR posts LGWR to do so.

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 usually writes fast enough to ensure that space is available in the buffer for new entries, even when access to the redo log is heavy.

A larger buffer makes it more likely that there is space for new entries, and also gives LGWR the opportunity to efficiently write out redo records (a too small log buffer on a system with large updates means that LGWR is continuously flushing redo to disk so that the log buffer remains 2/3 empty).

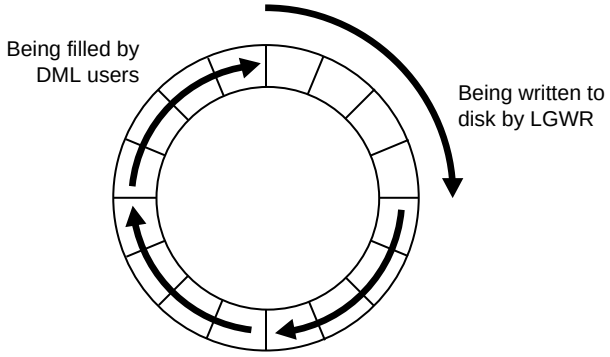
On machines with fast processors and relatively slow disks, the processors might be filling the rest of the buffer in the time it takes the redo log writer to move a portion of the buffer to disk. A larger log buffer may temporarily mask the effect of slower disks in this situation.

Good usage of the redo log buffer is a simple matter of:

- Batching commit operations for batch jobs, so that log writer is able to write redo log entries efficiently.
- Using NOLOGGING operations when you are loading large quantities of data

The size of the redo log buffer is determined by the initialization parameter LOG_BUFFER. The log buffer size cannot be modified after instance startup.

Figure 14–2 Redo Log Buffer



Sizing the Log Buffer

Applications that insert, modify, or delete large volumes of data usually need to change the default log buffer size. The log buffer is small compared with the total SGA size, and a modestly sized log buffer can significantly enhance throughput on systems that perform many updates.

A reasonable first estimate for such systems is to make the log buffer 1m. On most systems, sizing the log buffer larger than 1m does not provide any performance benefit. Increasing the log buffer size does not have any negative implications on performance or recoverability. It merely uses extra memory.

Log Buffer Statistics

The statistic REDO BUFFER ALLOCATION RETRIES reflects the number of times a user process waits for space in the redo log buffer. This statistic can be queried via the dynamic performance view V$SYSSTAT.

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 value of `redo buffer allocation retries` should be near zero over an interval. If this value increments consistently, then processes have had to wait for space in the redo log buffer. The wait can 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 is expressed in bytes. Alternatively, improve the checkpointing or archiving process.

Another data source is to check whether the `log buffer space` wait event is not a significant factor in the wait time for the instance; if not, the log buffer size is most likely adequate.

Configuring the PGA Working Memory

The Program Global Area (PGA) is a private memory region containing data and control information for a server process. Access to it is exclusive to that server process and is read and written only by the Oracle code acting on behalf of it. An example of such information is the runtime area of a cursor. Each time a cursor is executed, a new runtime area is created for that cursor in the PGA[1] memory region of the server process executing that cursor.

For complex queries (for example, decision support queries), a big portion of the runtime area is dedicated to work areas allocated by memory intensive operators, such as the following:

- Sort-based operators (for example, `ORDER BY`, `GROUP BY`, `ROLLUP`, window functions)
- Hash-join
- Bitmap merge
- Bitmap create
- Write buffers used by bulk load operations

A sort operator uses a work area (the sort area) to perform the in-memory sort of a set of rows. Similarly, a hash-join operator uses a work area (the hash area) to build a hash table from its left input.

The size of a work area *can be controlled and tuned*. Generally, bigger work areas can significantly improve the performance of a particular operator at the cost of higher memory consumption. Ideally, the size of a work area is big enough that it can accommodate the input data and auxiliary memory structures allocated by its

[1] Part of the run-time area can be located in the SGA when using shared server.

associated SQL operator. This is known as the *optimal* size of a work area. When the size of the work area is smaller than optimal, the response time increases, because an extra pass is performed over part of the input data. This is known as the *one-pass* size of the work area. Under the one-pass threshold, when the size of a work area is far too small compared to the input data size, multiple passes over the input data are needed. This could dramatically increase the response time of the operator. This is known as the *multi-pass* size of the work area. For example, a serial sort operation that needs to sort 10GB of data needs a little more than 10GB to run optimal and at least 40MB to run one-pass. If this sort gets less that 40MB, then it must perform several passes over the input data.

The goal is to have most work areas running with an optimal size (for example, more than 90% or even 100% for pure OLTP systems), while a smaller fraction of them are running with a one-pass size (for example, less than 10%). Multi-pass execution should be avoided. Even for DSS systems running large sorts and hash-joins, the memory requirement for the one-pass executions is relatively small. A system configured with a reasonable amount of PGA memory should not need to perform multiple passes over the input data.

Prior to release 9*i*, the maximum size of these working areas was controlled using the `SORT_AREA_SIZE`, `HASH_AREA_SIZE`, `BITMAP_MERGE_AREA_SIZE` and `CREATE_BITMAP_AREA_SIZE` parameters. Setting these parameters is difficult, because the maximum work area size is ideally selected based on the data input size and the total number of work areas active in the system. These two factors vary a lot from one work area to another and from one point in time to another. Thus, the various `*_AREA_SIZE` parameters are hard to tune under the best of circumstances.

See Also:

- *Oracle9i Data Warehousing Guide* for information on configuring the `HASH_AREA_SIZE`, `BITMAP_MERGE_AREA_SIZE` and `CREATE_BITMAP_AREA_SIZE` parameters
- "Configuring SORT_AREA_SIZE" on page 14-54

With release 9*i*, it is possible to simplify and improve the way the PGA is allocated. There is an automatic mode to dynamically adjust the size of the portion of the PGA memory dedicated to work areas. The size of that portion is adjusted based on an overall PGA memory target explicitly set by the DBA.

Note: This mechanism cannot be used for shared server connections.

Automatic PGA Memory Management

When running under the automatic PGA memory management mode, sizing of work areas for all *dedicated* sessions becomes automatic. Thus, the *_AREA_SIZE parameters are ignored by all sessions running in that mode. At any given time, the total amount of PGA memory available to work areas active in the instance is automatically derived from the parameter PGA_AGGREGATE_TARGET. This amount is set to the value of PGA_AGGREGATE_TARGET minus the PGA memory allocated by other components of the system (for example, PGA memory allocated by sessions). The resulting PGA memory is then assigned to individual active work areas based on their specific memory requirements.

The value of the PGA_AGGREGATE_TARGET initialization parameter, for example 100000K, 2500M, or 50G, should be set based on the total amount of memory available for the Oracle instance. This value can then be dynamically modified at the instance level.

For example, assume that an Oracle instance is configured to run on a system with 4GB of physical memory. Part of that memory should be left for the operating system and other non-Oracle applications running on the same hardware system. For example, the DBA might decide to dedicate only 80% of the available memory to the Oracle instance (3.2GB in the above example). The DBA must then decide how to divide the resulting memory between the SGA and the PGA. For OLTP systems, the PGA memory typically accounts for at most 20% of that total (that is, 80% for the SGA), while decision support systems (DSS) running large memory-intensive queries typically dedicate up to 70% of that memory to the PGA (that is, up to 2.2GB).

A good starting point to set the parameter PGA_AGGREGATE_TARGET could be:

- PGA_AGGREGATE_TARGET = (TOTAL_MEM * 80%) * 20% for an OLTP system
- PGA_AGGREGATE_TARGET = (TOTAL_MEM * 80%) * 50% for a DSS system

where TOTAL_MEM is the total amount of physical memory available on the system. Using the above computation and a value of TOTAL_MEM equal to 4GB, the DBA sets PGA_AGGREGATE_TARGET to 1600M for the DSS system and to 655M for the OLTP system.

After the PGA_AGGREGATE_TARGET parameter is configured and the instance is restarted, the PGA memory management of SQL work areas becomes fully automatic for all dedicated Oracle server processes. Under this automatic mode, Oracle tries to maximize the number of work areas that are using optimal memory and uses one-pass memory for the others. A work area should not run multi-pass under the automatic mode unless PGA_AGGREGATE_TARGET is set to a value that is

too small for the instance workload. If this is the case, then increase its value (and probably decrease the size of the SGA accordingly).

Several dynamic performance views help tune the value of the parameter PGA_AGGREGATE_TARGET:

- V$SYSSTAT and V$SESSTAT
- V$PGASTAT
- V$PROCESS
- V$SQL_WORKAREA_ACTIVE
- V$SQL_WORKAREA

See Also: *Oracle9i Database Reference*

V$SYSSTAT and V$SESSTAT

Statistics in the V$SYSSTAT and V$SESSTAT views show the total number of work areas executed with optimal memory size, one-pass memory size, and multi-pass memory size. These statistics are cumulative since the instance or the session was started.

The following query gives the total number and the percentage of times work areas were executed in these three modes since the instance was started:

```
SELECT name profile, cnt, decode(total, 0, 0, round(cnt*100/total)) percentage
    FROM (SELECT name, value cnt, (sum(value) over ()) total
    FROM V$SYSSTAT
    WHERE name like 'workarea exec%');
```

The output of this query might look like the following:

```
PROFILE                                  CNT PERCENTAGE
---------------------------------- ---------- ----------
workarea executions - optimal            5395         95
workarea executions - onepass             284          5
workarea executions - multipass             0          0
```

This output tells the DBA if he/she needs to change the setting of PGA_AGGREGATE_TARGET. This can be done dynamically with an ALTER SYSTEM statement. For example, if the number of multi-pass executions is not zero, then consider increasing the value of this parameter. Also consider increasing the value of this parameter when the percentage of one-pass executions is high compared to

optimal. On the other hand, if the percentage of optimal work area is 100%, then consider reducing the PGA target.

V$PGASTAT

The V$PGASTAT view gives instance level statistics on the PGA usage and the automatic memory manager. All values are returned in bytes. For example:

```
SELECT name, value
FROM V$PGASTAT;
```

The output of this query might look like the following:

```
NAME                                                             VALUE
---------------------------------------------------------------- ----------
aggregate PGA auto target                                        912052224
global memory bound                                                  51200
total expected memory                                               262144
total PGA inuse                                                   35444736
total PGA allocated                                               53967872
maximum PGA allocated                                             66970624
total PGA used for auto workareas                                   262144
maximum PGA used for auto workareas                                5292032
total PGA used for manual workareas                                      0
maximum PGA used for manual workareas                                    0
estimated PGA memory for optimal                                    262144
maximum PGA memory for optimal                                   500123520
estimated PGA memory for one-pass                                   262144
maximum PGA memory for one-pass                                   70123520
```

The main statistics are:

- `aggregate PGA auto target`: This gives the amount of PGA memory Oracle can use for work areas running in automatic mode. This amount is derived from the value of the parameter PGA_AGGREGATE_TARGET. If this value is too low compared to PGA_AGGREGATE_TARGET, then a lot of PGA memory is used by other components of the system (for example, PL/SQL or Java memory) and little is left for work areas.
- `global memory bound`: This gives the maximum size of a work area executed in AUTO mode. This value is continuously adjusted by Oracle to reflect to current state of the work area workload. It generally decreases when the number of active work areas is increasing in the system. In general, the value of the global bound should not decrease below one megabyte. If it does, then the value of PGA_AGGREGATE_TARGET should probably be increased.

- `total PGA allocated`: This gives the current amount of PGA memory allocated by the instance. Oracle tries to keep this number below the value of PGA_AGGREGATE_TARGET. However, it is possible for the PGA to be over allocated by a small amount and for a short period of time when the work area workload is increasing very rapidly.
- `total PGA used for auto workareas`: This indicates how much PGA memory is consumed by work areas running in automatic memory management mode. This number can be used to determine how much memory is consumed by other consumers of PGA memory (for example, PL/SQL or Java):

  PGA other = '`total PGA allocated`' - '`total PGA used for auto workareas`'

  Over allocating PGA memory can also happen if the value of PGA_AGGREGATE_TARGET is too small to accommodate this 'PGA other' component. If this is the case (that is, 'PGA other' > PGA_AGGREGATE_TARGET), then increase the value of PGA_AGGREGATE_TARGET.
- `estimated PGA memory for optimal/one-pass`: This estimates how much PGA memory is required to run all active work areas in optimal mode (respectively in one-pass mode). To avoid running one or more work areas in multi-pass mode, ensure that PGA_AGGREGATE_TARGET is set to at least the **maximum** value of the estimated PGA memory for one-pass and generally to a higher value such that some work areas get enough memory to run in optimal mode.

V$PROCESS

The V$PROCESS view has one row per Oracle process connected to the instance. The columns PGA_USED_MEM, PGA_ALLOC_MEM and PGA_MAX_MEM can be used to monitor the PGA memory usage of these processes. For example:

```
SELECT program, pga_used_mem, pga_alloc_mem, pga_max_mem
FROM V$PROCESS;
```

The output of this query might look like the following:

```
PROGRAM                        PGA_USED_MEM PGA_ALLOC_MEM PGA_MAX_MEM
------------------------------ ------------ ------------- -----------
PSEUDO                                    0             0           0
oracle@miflo (PMON)                  120463        234291      234291
oracle@miflo (DBW0)                 1307179       1817295     1817295
oracle@miflo (LGWR)                 4343655       4849203     4849203
oracle@miflo (CKPT)                  194999        332583      332583
oracle@miflo (SMON)                  179923        775311      775323
oracle@miflo (RECO)                  129719        242803      242803
oracle@miflo (TNS V1-V3)            1400543       1540627     1540915
oracle@miflo (P000)                  299599        373791      635959
oracle@miflo (P001)                  299599        373791      636007
oracle@miflo (P002)                  299599        373791      570471
oracle@miflo (P003)                  303899        373791      636007
oracle@miflo (P004)                  299599        373791      635959
```

V$SQL_WORKAREA_ACTIVE

Query V$SQL_WORKAREA_ACTIVE to display the work areas that are active (or executing) in the instance. Small active sorts (under 64KB) are excluded from the view. Use this view to precisely monitor the size of all active work areas. For example:

```
SELECT   to_number(decode(SID, 65535, NULL, SID)) sid,
         operation_type OPERATION,
         trunc(WORK_AREA_SIZE/1024) WSIZE,
         trunc(EXPECTED_SIZE/1024) ESIZE,
         trunc(ACTUAL_MEM_USED/1024) MEM,
         trunc(MAX_MEM_USED/1024) "MAX MEM",
         NUMBER_PASSES PASS
FROM V$SQL_WORKAREA_ACTIVE
ORDER BY 1,2;
```

The output of this query might look like the following[1]:

```
SID OPERATION             WSIZE     ESIZE       MEM   MAX MEM PASS
--- -------------------- ----- --------- --------- --------- ----
 27 GROUP BY (SORT)          73        73        64        64    0
 44 HASH-JOIN              3148      3147      2437      2437    0
 71 HASH-JOIN             13241     19200     12884     34684    1
```

This output shows that session 71 is running a hash-join whose work area is running in one-pass mode. This work area is currently using 12MB of memory (MEM column) and has used in the past up to 34MB of PGA memory (MAX MEM column).

[1] The memory sizes has been converted into kilobytes by dividing by 1024.

When a work area is deallocated, that is, when the execution of its associated SQL operator is completed, the work area is automatically removed from the V$SQL_WORKAREA_ACTIVE view.

V$SQL_WORKAREA

Oracle maintains cumulative work area statistics for each loaded cursor whose execution plan uses one or more work areas. Every time a work area is deallocated, the V$SQL_WORKAREA table is updated with execution statistics for that work area.

V$SQL_WORKAREA can be joined with V$SQL to relate a work area to a cursor. It can even be joined to V$SQL_PLAN to precisely determine which operator in the plan uses a work area.

The following example finds the top 10 work areas requiring most cache memory:

```
SELECT *
FROM
     ( SELECT workarea_address, operation_type, policy, estimated_optimal_size
       FROM V$SQL_WORKAREA
       ORDER BY estimated_optimal_size )
WHERE ROWNUM <= 10;
```

The following example finds the cursors with one or more work areas that have been executed in one or even multiple passes:

```
SELECT sql_text, sum(ONEPASS_EXECUTIONS), sum(MULTIPASSES_EXECUTIONS)
FROM V$SQL s, V$SQL_WORKAREA wa
WHERE s.address = wa.address
GROUP BY sql_text
HAVING sum(ONEPASS_EXECUTIONS+MULTIPASSES_EXECUTIONS)>0;
```

For the hash value and address of a particular cursor, the following query displays the cursor execution plan, including information about the associated work areas.

```
SELECT operation, options, object_name name,
       trunc(bytes/1024/1024) "input(MB)",
       trunc(last_memory_used/1024) last_mem,
       trunc(estimated_optimal_size/1024) optimal_mem,
       trunc(estimated_onepass_size/1024) onepass_mem,
       decode(optimal_executions, null, null,
              optimal_executions||'/'||onepass_executions||'/'||
              multipasses_exections) "O/1/M"
FROM V$SQL_PLAN p, V$SQL_WORKAREA w
WHERE p.address=w.address(+)
  AND p.hash_value=w.hash_value(+)
  AND p.id=w.operation_id(+)
  AND p.address='88BB460C';

OPERATION    OPTIONS  NAME     input(MB) LAST_MEM OPTIMAL_ME ONEPASS_ME O/1/M
------------ -------- -------- --------- -------- ---------- ---------- ------
SELECT STATE
SORT         GROUP BY              4582        8         16         16 16/0/0
HASH JOIN    SEMI                  4582     5976       5194       2187 16/0/0
TABLE ACCESS FULL     ORDERS         51
TABLE ACCESS FUL      LINEITEM     1000
```

The address and hash value can be obtained from the V$SQL view given a pattern on the SQL text of the query of interest. For example:

```
SELECT address, hash_value
FROM V$SQL
WHERE sql_text LIKE '%my pattern%';
```

Configuring SORT_AREA_SIZE

Tuning sort operations using SORT_AREA_SIZE is only relevant for configurations running the Oracle shared server option or for configurations *not* running under the automatic memory management mode. In the later case, Oracle Corporation strongly recommends switching to the automatic memory management mode, because it is easier to manage and often outperforms a manually-tuned system.

This section describes the following:

- Fundamentals of Sorts
- Recognizing Memory and Disk Sorts
- Application Characteristics
- Considerations with SORT_AREA_SIZE
- Considerations with SORT_AREA_RETAINED_SIZE
- Using NOSORT to Create Indexes Without Sorting
- Using GROUP BY NOSORT

Fundamentals of Sorts

A sort is an operation that orders data according to certain criteria before the data is returned to the requestor.

Operations that perform sorts include the following:

- `CREATE INDEX`
- `SELECT .... ORDER BY`
- `SELECT DISTINCT`
- `SELECT .... GROUP BY`
- `SELECT ... CONNECT BY`
- `SELECT ... CONNECT BY ROLLUP`
- Sort merge joins

> **See Also:** *Oracle9i Database Concepts* for a list of SQL statements that perform sorts

When the `WORKAREA_SIZE_POLICY` parameter is set to `MANUAL`, the maximum amount of memory allocated for a sort is defined by the parameter `SORT_AREA_SIZE`. If the sort operation is not able to completely fit into `SORT_AREA_SIZE` memory, then the sort is separated into phases. The temporary output of each phase is stored in temporary segments on disk. The tablespace in which these sort segments are created is the user's temporary tablespace.

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 that the sort can merge more runs in one merge pass.

Note: Reads and writes performed as a part of disk sort operations bypass the buffer cache.

Recognizing Memory and Disk Sorts

Oracle collects statistics that reflect sort activity and stores them in dynamic performance views, such as V$SQLAREA and V$SYSSTAT.

The following statistics from V$SYSSTAT reflect sort behavior:

`sorts (memory)`	The number of sorts small enough to be performed entirely in memory without I/O to temporary sort segments on disk.
`sorts (disk)`	The number of sorts too large to be performed entirely in memory, requiring I/O to temporary sort segments on disk.

For example, the following query monitors these statistics:

```
SELECT NAME, VALUE
  FROM V$SYSSTAT
 WHERE NAME IN ('sorts (memory)', 'sorts (disk)');
```

The output of this query might look like the following:

```
NAME                                                                  VALUE
---------------------------------------------------------------- ----------
sorts (memory)                                                    430418566
sorts (disk)                                                          33255
```

In addition, to find individual SQL statements that are performing sorts, query the V$SQLAREA view. Order the rows by SORTS to identify the SQL statements performing the most sorts. For example:

```
SELECT HASH_VALUE, SQL_TEXT, SORTS, EXECUTIONS
  FROM V$SQLAREA
 ORDER BY SORTS;
```

In an OLTP environment, the best solution is to investigate whether the SQL statements can be tuned to avoid the sort activity.

Application Characteristics

For best performance in OLTP systems, most sorts should occur solely within memory. Sorts written to disk can adversely affect performance. If your OLTP application frequently performs sorts that do not fit into sort area size, and if the application has been tuned to avoid unnecessary sorting, then consider increasing the SORT_AREA_SIZE parameter for the whole instance.

If there are only a few programs that perform larger than average sorts that sort to disk, then it is possible to modify SORT_AREA_SIZE at the session level only for that workload or application (for example, building an index).

DSS applications typically access large volumes of data. These types of applications are expected to perform sorts to disk, purely because of the nature of the application and the data volumes involved. In DSS applications, it is important to identify the optimal SORT_AREA_SIZE to allow the disk sorts to perform most efficiently. Allocating more memory to sorts does not necessarily mean that the sort will be faster.

Considerations with SORT_AREA_SIZE

The main consideration when choosing SORT_AREA_SIZE is balancing memory usage with sort performance.

Since Oracle8 release 8.0, sorts do not allocate the whole of SORT_AREA_SIZE in one memory allocation at the beginning of the sort. The memory is allocated in DB_BLOCK_SIZE chunks when required, up to SORT_AREA_SIZE.

This means that increasing SORT_AREA_SIZE memory is a concern when the majority of processes on the system perform sorts *and* use the maximum allocation. In this situation, increasing SORT_AREA_SIZE for the instance as a whole results in more memory being allocated from the operating system (for dedicated connections; that is, if a shared server environment is not used). This is not necessarily a problem if the system has free memory available. However, if there is not enough free memory, then this causes paging and/or swapping.

If a shared server environment is used, then the additional memory is allocated out of the shared pool (the large pool in the shared pool, if this is configured). The actual amount of memory used in the shared pool is the lesser of SORT_AREA_SIZE, SORT_AREA_RETAINED_SIZE, and the actual allocation used by the sort.

If the SORT_AREA_SIZE is too small, then the sort is not performed as efficiently as possible. This means that sorts that could have been memory-only sorts will be disk sorts, or, alternatively, that the number of sort runs required to process the sort

could be larger than necessary. Both of these situations can severely degrade performance.

Remember that there is a point after which increasing the SORT_AREA_SIZE no longer provides a performance benefit.

Increasing SORT_AREA_SIZE 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 might benefit from increasing the value of SORT_AREA_SIZE. Alternatively in a DSS environment, increasing SORT_AREA_SIZE is not likely to make the sort a memory-only sort; however, depending on the current value and the new value chosen, it could make the sort faster.

The maximum value of this parameter depends on your operating system. You need to determine what size SORT_AREA_SIZE makes sense for your system.

Considerations with SORT_AREA_RETAINED_SIZE

The SORT_AREA_RETAINED_SIZE parameter determines the lower memory limit to which 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.

With dedicated connections, the freed memory is *not* released to the operating system, rather *the freed memory is made available to the session* for reuse.

However, if the connection is via shared server, then there could be a memory benefit to setting SORT_AREA_RETAINED_SIZE. If this parameter is set after the sort has completed, then the sorted data is stored in the SGA. The amount of memory used in the SGA is the lesser of the actual usage or SORT_AREA_RETAINED_SIZE if it is set; otherwise, it is SORT_AREA_SIZE. This is why setting SORT_AREA_RETAINED_SIZE could be of use with a shared server environment.

> **Note:** Connections made to the database through shared servers usually should not perform large sorts.

Although there might be a memory saving with shared server, setting SORT_AREA_RETAINED_SIZE causes additional I/O to write and read data to and from temporary segments on disk (if the sort requires more than SORT_AREA_RETAINED_SIZE bytes).

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. However, Oracle lets you 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 might 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, the following 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 might not be the fastest way to load a table.

- If you have a multiple-CPU computer, then you might 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 sort your data before loading, if possible. Then, create the index *with* the NOSORT clause.

Using 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 can 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.

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.

If the application is a large OLTP system, dramatic reductions in server memory usage can 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 shared server architecture is an effective solution for reducing the number of server operating system processes. Shared server is also quite effective at reducing overall memory requirements. You can also use shared server to reduce the number of network connections when you use a shared server architecture with connection pooling and session multiplexing.

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.

15

I/O Configuration and Design

The input/output (I/O) subsystem is a vital component of an Oracle database. This chapter introduces fundamental I/O concepts, discusses the I/O requirements of different parts of the database, and provides sample configurations for I/O subsystem design.

This chapter includes the following topics:

- Understanding I/O
- Basic I/O Configuration

Understanding I/O

The performance of many software applications is inherently limited by disk I/O. Applications that spend the majority of CPU time waiting for I/O activity to complete are said to be *I/O bound*.

Oracle is designed so that a well written application's performance should not be limited by I/O. Tuning I/O can enhance the performance of the application if the I/O system is operating at capacity and is not able to service the I/O requests within an acceptable time. However, tuning I/O cannot help performance if the application is not I/O bound (for example, when CPU is the limiting factor).

Designing I/O Layouts

Consider the following database requirements when designing an I/O system:

1. Storage, such as minimum bytes of disk
2. Availability, such as 24x7, 9x5
3. Performance, such as I/O throughput and application response times

Many I/O designs plan for storage and availability requirements with the assumption that performance will not be an issue. This is not always the case. Optimally, the number of disks and controllers should be determined by I/O throughput and redundancy requirements. Then, the size of disks can be determined by the storage requirements.

Disk Performance and Reliability

For any database, the I/O subsystem is critical for system availability, performance, and data integrity. A weakness in any of these areas can render the database system unstable, unscalable, or untrustworthy.

All I/O subsystems use magnetic disk drives. Conventional magnetic disk drives contain moving parts. Because these moving parts have a design life, they are subject to tolerances in manufacturing that make their reliability and performance inconsistent. Not all theoretically-identical drives perform the same, and they can break down over time. When assembling a large disk configuration, you need only look at the mean time between failures of disk drives and the number of disks to see that disk failures are a very common occurrence. This is unfortunate, because the core assets of any system (the data) reside on the disks.

Disk Technology

The main component of any I/O subsystem, the disk drive, has barely changed over the last few years. The only changes are the increase in capacity of the drive from under one Gigabyte to over 50 Gigabytes, and small improvements in disk access times, and hence throughput. This is very different from the performance improvements made in CPUs, which have doubled their performance every 18 months.

There is a disk drive paradox that says that if you size the number of disks required by disk capacity, then you need fewer and fewer disks over time as they increase in size. Or, if you size the number of disks by performance, then you must double the number of disks per CPU every 18 months. Dealing with this paradox has proved difficult to many system configurations, especially those wanting to make systems cheaper by using less disks.

In addition to the size of the actual disks manufactured, the way disk subsystems are connected has changed. On smaller systems, in general, disk drives are connected individually to the host machine by SCSI interfaces through one of a number of disk controllers. For high-end systems, disk mirroring and striping are essential for performance and availability. These requirements lead to hundreds of disks connected through complex wiring configurations. Also, this type of configuration can have poor fault resilience, because it is difficult to hot swap broken components.

However, although disk technology itself has not greatly changed, I/O subsystems have evolved into disk arrays that overcome many of the described problems. The systems perform disk mirroring, provide hot swapping of disks, and in many cases, provide simpler connections to the host by fiber interfaces. The more sophisticated disk arrays are in fact small computers themselves with their own battery backed memory cache for high performance resilient writes, dial home diagnostics, and proxy backup to allow backup without going through the host operating system.

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 must wait to access the disk.

Load Balancing and Striping

A performance engineer's goal is to distribute the I/O load evenly across the available devices. This is known as *load balancing* or *distributing I/O*. Historically,

load balancing had to be performed manually. DBAs would determine the I/O requirements and characteristics of each datafile and design the I/O layout based on which files could be placed together on a single disk to distribute the activity evenly over all disks.

If a particular table or index was very I/O-intensive, then to further distribute the I/O load, DBAs also had the option of manually distributing (*striping*) the data. This was achieved by splitting the object's extents over separate datafiles, then distributing the datafiles over devices.

Fundamentally, striping divides data into small portions and stores these portions in separate files on separate disks. This allows multiple processes to access different portions of the data concurrently without disk contention. Operating systems, hardware vendors, and third party software vendors provide the tools to be able to stripe a heavily-used file across many physical devices simply, thus making the job of balancing I/O significantly easier for DBAs.

Striping is helpful both in OLTP environments to optimize random access to tables with many rows and also in DSS environments to allow parallel operations to scan large volumes of data quickly. Striping techniques are productive when the load redistribution eliminates or reduces some form of queue. If the concurrent load is too heavy for the available hardware, then striping does not alleviate the problem.

Striping and RAID

You must also consider the recoverability requirements of each particular system. Redundant arrays of inexpensive disks (RAID) configurations provide improved data reliability while offering the option of striping. The RAID level chosen should depend on performance and cost. Different RAID levels are suited to different types of applications, depending on their I/O characteristics.

Following are brief overviews of the most popular RAID configurations used for database files, along with applications that best suit them. These descriptions are very general. Consult with your hardware and software vendors for specific details on their implementations:

- RAID 0: Striping. Files are striped across many physical disks. This provides read and write performance, but not reliability.

  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.

- RAID 1: Mirroring. Physical disks are used to store *N* concurrently maintained copies of a file. The number of disks required is N times M, where M is the number of disks required to store the original files. RAID 1 provides good reliability and good read rates. Sometimes, writes can be costly, because N writes are required to maintain N copies.
- RAID 0+1: Striping and mirroring. This level combines the technologies of RAID 0 and RAID 1. It is widely used because it provides good reliability and better read and write performance than RAID 1.

  Note: Mirroring can cause I/O bottlenecks. Generally, the process of writing to each mirror is done in parallel and does not cause a bottleneck. However, if each mirror is striped differently, then the I/O does not complete until the slowest member of the mirror is complete. To avoid I/O problems, stripe the copies using the same number of disks as for the primary database.

- RAID 5: Striping and redundancy. RAID 5 striping is similar to striping in RAID 0. Recoverability of lost data due to disk failure is achieved by storing parity data regarding all disks in a stripe, uniformly throughout disks in the group. Compared to RAID 1, the benefit is the saving in disk cost. RAID 5 provides good reliability. Sequential reads benefit the most, while write performance can suffer. This configuration might not be ideal for write-intensive applications. However, some implementations of RAID 5 provided today avoid many of the traditional RAID 5 limits.

Balancing Budget, Performance, and Availability

The choice of I/O subsystem and how to design the layout is a compromise of budget, performance, and availability. It is essential to be able to expand the I/O subsystem as throughput requirements grow and more I/O is performed. To achieve a scalable I/O system, the chosen system must be able to evolve over time, with a minimum of downtime. In many cases, this involves configuring the system to achieve 95% optimal performance, forsaking 5% performance for ease of configuration and eventual upgrades.

Basic I/O Configuration

This section describes the basic information to be gathered, and decisions to be made when defining your system's I/O configuration. You want to keep the configuration as simple as possible, while maintaining the required availability,

recoverability and performance. The more complex a configuration becomes, the more difficult it is to administer, maintain, and tune.

Determining Application I/O Characteristics

This section describes the basic I/O characteristics and requirements that must be determined. These I/O characteristics influence the decisions on what type of technology is required and how to configure that technology. I/O requirements include the performance, space, and recoverability needs specific to the site. This information must be determined order to design an efficient I/O system:

- Read Rate and Write Rate
- I/O Concurrency
- I/O Size
- Availability
- Storage Size

Read Rate and Write Rate

The *read rate* is the number of reads per second. The *write rate* is the number of writes per second. The sum of the read rate and write rate is the I/O rate (the number of I/O operations per second). For a well performing system, your application's I/O rate should be a significant factor when determining the *absolute minimum* number of disks, controllers, and so on required.

A system that has a high write rate might also benefit from the following configuration options:

- Using raw devices, or third party software that is able to perform reads directly to disk, avoiding reading to and writing from the Unix buffer cache.
- Using an I/O system that has a large-enough disk cache to sustain destaging dirty blocks without the cache filling up.
- Avoiding RAID 5 configuration

> **See Also:** Your vendor-specific documentation to see whether high-write rates can be sustained without a performance hit

I/O Concurrency

I/O concurrency measures the number of distinct processes simultaneously having I/O requests to the I/O system outstanding. From an Oracle perspective, I/O

concurrency is considered the number of processes concurrently issuing I/O requests. A high degree of I/O concurrency implies that there are many distinct processes simultaneously issuing I/O requests. A low degree of concurrency implies that few processes are simultaneously issuing I/O requests.

I/O Size

I/O size is the size of the I/O request from Oracle's perspective. This ranges from the minimum, being the operating system block size, to the maximum, typically a factor of the Oracle block size multiplied by the multiblock read count.

Although the I/O size can vary depending on the type of operation, there are some reasonable, general estimates that can be made depending on the nature of the application.

- With a DSS system, the majority of the I/Os are typically going to be large, in the order of N * `DB_BLOCK_SIZE`.
- With an OLTP system, the I/Os are predominantly going to be of size `DB_BLOCK_SIZE`.

Factors Affecting I/O Size and Concurrency In an Oracle system, the various files have different requirements of the disk subsystem. The requirements are distinguished by the rate of reading and writing of data, and by the concurrency of these operations. High rates are relatively easily sustained by the disk subsystem when the concurrency is low, and vice versa. It is important that your design of the disk subsystem take the following factors into consideration:

- Operational parameters

  The configurable and nonmodifiable operational parameters for Oracle, the operating system, the hardware, and any third party software (for example, LVM).

  See Also: Table 15–1 on page 15-11

- The typical I/O size and degree of concurrency for each file

  The typical I/O size and degree of concurrency for each of the various Oracle files varies considerably from one site to another, even for the same type of file. For example, a site that performs many concurrent disk sorts has different characteristics for its TEMP tablespace than a site that performs predominantly small, in-memory sorts.

The concurrency and type of I/O (read/write) is provided below for a theoretical installation. The example describes a high-query, high-update OLTP application that performs predominantly in-memory sorts, and whose indexes remain cached in the buffer cache.

Component	Read rate	Write rate	Concurrency
Archive logs	High	High	Low
Redo logs	High	High	Low
Rollback segment tablespaces	Low	High	High
TEMP tablespaces	Low	Low	High
Index tablespaces	Low	Medium	High
Data tablespaces	High	Medium	High
Application log and output files	Low	Medium	High
Binaries (shared)	Low	Low	High

Availability

Site-specific availability requirements impose additional specifications on the disk storage technology and layout. Typical considerations include the appropriate RAID technology to suit the recoverability requirements and any Oracle-specific safety measures, such as mirroring of redo logs and archivelogs and mirroring of control files.

Dedicating separate disks to mirroring redo log files is an important safety precaution. This ensures that the datafiles and the redo log files cannot both be lost in a single disk failure.

> **See Also:** *Oracle9i User-Managed Backup and Recovery Guide* for more information on recoverability

Storage Size

With the large disks available today, if the performance and availability requirements are satisfied, then the storage needs in most cases have already been met. If, however, the system stores voluminous data online, which does not have high concurrency or throughput requirements, then you might fall short of storage requirements. In such cases, to maximize storage space, consider the following:

- Use different RAID configurations.
- Use more disks.

- Use larger disks.

I/O Configuration Decisions

Depending on the available hardware and software, decisions to be made during the I/O layout phase include determining the following:

- The number of disks available for database files
- The RAID level
- The stripe unit (stripe size) and the stripe width (number of disks)
- Using raw devices or file system
- Whether the system can perform asynchronous I/O

Goal - Distributing I/O

To maximize performance, the goal is to distribute the database's I/O load as evenly as possible over the available disks.

In addition to distributing I/O, other considerations when designing a system's I/O layout should include the expected growth on the system and the required recoverabilty.

Know your I/O System

You should be aware of the capabilities of your I/O system. This information is available by reviewing your hardware and software documentation (if applicable), and also by performing tests at your site. Tests can include varying any of the following factors:

- I/O sizes
- Raw devices compared to file systems
- Read performance, write performance
- Sequential and random access
- Configuring asynchronous I/O compared to synchronous I/O

Performing this type of research typically also provides insight on potential configurations that can be used for the final design.

Match I/O Requirements with the I/O System

If you have a benchmark that simulates the load placed on the I/O system by the application, or if there is an existing production system, then look at the Oracle statistics to determine the I/O rates per file and for the database as a whole.

> **See Also:** *Oracle9i Database Performance Methods* for more information on operating system I/O statistics

To determine the Oracle file I/O statistics, look at the following:

- The number of physical reads (V$FILESTAT.PHYRDS)
- The number of physical writes (V$FILESTAT.PHYWRTS)
- The average read time
- I/Os = physical reads + physical writes.

  The sum of the physical reads and physical writes gives the number of physical I/Os from Oracle's perspective.

Assuming that the Oracle buffer cache is adequately sized, the physical reads and physical writes statistics are useful.

Another important derivable statistic is the average number of I/Os per second. Sample the V$FILESTAT data over an interval, add physical reads and writes, then divide this by the elapsed time in seconds to determine this ratio. The general formula is as follows:

Avg I/Os=(change in physical reads + change in physical writes)/elapsed seconds

> **See Also:** Chapter 24, "Dynamic Performance Views for Tuning" for details on how to calculate the delta information in V$FILESTAT

To estimate the I/O requirements, scale this data with the expected workload on the new system. Comparing the scaled data with the disk capabilities can potentially identify whether there will be a mis-match between the new application's I/O requirements and the capabilities of the I/O system.

Also identify any I/O-intensive operations that are not part of the typical load, and whose I/O rates would greatly peak above the average. Ensure that your system is able to sustain these rates. Operations such as index builds, data loads, and batch processing fall into this category.

See Also: Chapter 24, "Dynamic Performance Views for Tuning" for details on how to calculate the delta information in V$FILESTAT

Layout the Files using Operating System or Hardware Striping

If your operating system has logical volume manager (LVM) software or hardware-based striping, then it is possible to distribute I/O using these tools. Decisions to be made when using an LVM or hardware striping include the stripe size and the stripe width. Choose these values wisely, so that the system is capable of sustaining the required throughput.

Different types of applications benefit from different stripe sizes. The optimal stripe size and stripe width is dependent on the following:

- Requested I/O Size
- Concurrency of I/O Requests
- Alignment of Physical Stripe Boundaries with Block Size Boundaries
- Manageability of the Proposed System

Requested I/O Size

The size of an I/O is affected by the following Oracle and operating system (OS) operational parameters:

Table 15–1 Oracle and Operating System Operational Parameters

DB_BLOCK_SIZE	The size of single-block I/O requests. This parameter is also used in combination with multiblock parameters to determine multiblock I/O request size.
OS block size	Determines I/O size for redo log and archive log operations.
Maximum OS I/O size	Places an upper bound on the size of a single I/O request.
DB_FILE_MULTIBLOCK_READ_COUNT	The maximum I/O size for full table scans is computed by multiplying this parameter with DB_BLOCK_SIZE. (the upper value is subject to operating system limits).
SORT_AREA_SIZE	Determines I/O sizes and concurrency for sort operations.
HASH_AREA_SIZE	Determines the I/O size for hash operations.

Along with I/O size, also consider the following when choosing stripe width and striping unit:

- On low concurrency systems, ensure that no single I/O visits the same disk twice. For example, assume that the stripe width is four disks, and the stripe unit is 32k. If a single 1MB I/O request (for example, for a full table scan) was issued by an Oracle server process, then each disk in the stripe would need to perform four I/Os to return the requested data. To avoid this situation, the size of the average I/O should be smaller than the stripe width multiplied by the stripe size. If this is not the case, then a single I/O request made by Oracle to the operating system results in multiple physical I/O requests to the same disk.
- On high concurrency systems, ensure that no single I/O request is 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 can severely degrade the I/O response times.

Concurrency of I/O Requests

- In a system where there is a high degree of concurrent small I/O requests, such as in a traditional OLTP environment, it is beneficial to keep the stripe size large. Using stripe sizes larger than the I/O size is called **coarse grain striping**. In an OLTP system, the stripe size is N * `DB_BLOCK_SIZE` (where N is greater than 1). This allows a single I/O request to be serviced by a single disk, and hence a large number of concurrent I/O requests to be serviced by a set of striped disks. Parallel query in a DSS environment is also a candidate for coarse grained striping. This is because there are many individual processes each issuing separate I/Os.
- In a system where there are few large I/O requests, such as in a traditional DSS environment, it is beneficial to keep the stripe size small. This is called **fine grain striping**. For example the stripe size is N * `DB_BLOCK_SIZE`, where N is smaller than the multiblock read parameters (such as `DB_FILE_MULTIBLOCK_READ_COUNT`). This allows a single I/O request to be serviced by multiple disks. The I/O requests should be large enough so that the data transfer time is a significant part of the service time (opposed to the call set up time).

Alignment of Physical Stripe Boundaries with Block Size Boundaries

On some Oracle ports, it is possible that an Oracle block boundary will not align with the stripe. If your stripe size is the same size as the Oracle block, then a single I/O issued by Oracle might result in two physical I/O operations.

See Also: Your port-specific documentation for your platform

This is not optimal in an OLTP environment. To ensure a higher probability of one logical I/O resulting in no more than one physical I/O, the minimum stripe size should be at least twice the Oracle block size.

Table 15–2 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` multiplied by the Oracle block size

Manageability of the Proposed System

With an LVM, the simplest possible configuration is to create a single striped volume over all available disks. In this case, the stripe width encompasses all available disks. All database files reside within that volume, effectively distributing the load evenly. This single-volume layout provides adequate performance in most situations.

A 'single-volume' configuration is only viable when used in conjunction with RAID technology that allows easy recoverability, such as RAID 1. Otherwise, losing a single disk means losing all files concurrently, and hence a full database restore and recovery.

In addition to performance, there is a manageability concern: the design of the system must allow disks to be added simply, to allow for database growth. The challenge is to do so while keeping the load balanced evenly.

For example, an initial configuration can involve the creation of a single striped volume over 64 disks, each disk being 16GB. This is total disk space of 1Tb for the primary data. Sometime after the system is operational, an additional 80 GB (that is, five disks) must be added to account for future database growth.

The options for making this space available to the database include creating a second volume that includes the five new disks. However, an I/O bottleneck develops if these new disks are unable to sustain the I/O throughput required for the files placed on them.

Another option is to increase the size of the original volume. LVMs are becoming sophisticated enough to allow dynamic reconfiguration of the stripe width, which allows disks to be added while the system is online. This begins to make the placement of all files on a single striped volume feasible in a production environment.

If your LVM is unable to support dynamically adding disks to the stripe, then it is likely that you need to choose a smaller manageable stripe width. This allows growing the system by a stripe width when new disks are added.

In the example above, it is possible a more 'managable' stripe width is eight disks. This is only feasible if eight disks are capable of sustaining the required number of I/Os per second. Thus, when extra disk space is required, another eight-disk stripe can be added, keeping the I/O balanced across the volumes.

Note: The smaller the stripe width becomes, the more likely it is that you need to spend time distributing the files on the volumes, and the closer the procedure becomes to manually distributing I/O.

Manually Distributing I/O

If your system does not have an LVM or hardware striping, then I/O must be manually balanced across the available disks by distributing the files according to each file's I/O requirements. In order to make decisions on file placement, you should be familiar with the I/O requirements of the database files and the capabilities of the I/O system. If you are not familiar with this data and do not have a representative work-load to analyze, then expect to make a first guess, and then tune the layout as the usage becomes known.

To stripe disks manually, you need to relate a file's storage requirements to its I/O requirements.

1. Begin by evaluating a database's disk storage requirements by checking the size of the files and the size of the disks.
2. Identify the expected I/O throughput per file. Determine which files have the highest I/O rate and which do not have many I/Os. Lay out the files on all the available disks so as to even out the I/O rate.

One popular theory suggests separating a frequently used table from its index. This is not correct. 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. It is not sufficient to separate a datafile simply because the datafile contains indexes or table data - the decision to segregate a file should only be made when the I/O rate for that file affects database performance.

When to Separate Files

Regardless of whether operating system striping or manual I/O distribution is used, if your I/O system or I/O layout is not able to support the I/O rate required, then you need to separate high I/O files from the remaining files. This can be identified either at the planning stage or after the system is live.

The decision to segregate files should only be driven by I/O rates, recoverability concerns, or manageability issues. (For example, if your LVM does not support dynamic reconfiguration of stripe width, then you might need to create smaller stripe widths to be able to add *N* disks at time to create a new stripe of identical configuration.)

Before segregating files, verify that the bottleneck is truly an I/O issue. The data produced from investigating the bottleneck identifies which files have the highest I/O rates.

See Also:

- "Identifying and Gathering Data on Resource-Intensive SQL" on page 6-3

Tables, Indexes, and TEMP Tablespaces

If the high I/O files are datafiles belonging to tablespaces that contain tables and indexes, then identify whether the I/O for those files can be reduced by tuning SQL or application code.

If the high-I/O files are datafiles that belong to the TEMP tablespace, then investigate whether it is possible to tune the SQL statements performing disk sorts to avoid this activity, or tune the sorting.

After the application has been tuned to avoid unnecessary I/O, if the I/O layout is still not able to sustain the required throughput, then consider segregating the high I/O files.

See Also: "Identifying and Gathering Data on Resource-Intensive SQL" on page 6-3

Redo Log files

If the high-I/O files are redo log files, then consider splitting the redo log files from the other files. Possible configurations can include the following:

- Place all redo logs on one disk without any other files. Also consider availability: members of the same group should be on different physical disks and controllers for recoverability purposes.
- Place each redo log group on a separate disk that does not store any other files.
- Stripe the redo log files across several disks using an operating system striping tool. (Manual striping is not possible in this situation.)
- Avoid using RAID 5 for redo logs.

Redo log files are written by the Log Writer process sequentially. This operation can be made 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 necessary. If your system supports asynchronous I/O but this feature is not currently configured, then test to see if using this feature is beneficial. Performance bottlenecks related to LGWR are rare.

Archived Redo Logs

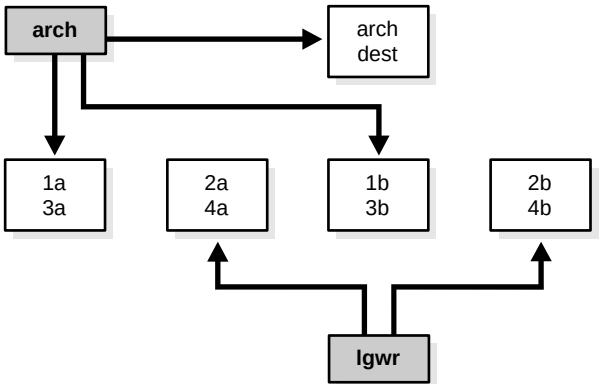
If the archiver is slow, then it might be prudent to prevent I/O contention between the archiver process and LGWR by ensuring that archiver reads and LGWR writes are separated.This is achieved by placing logs on alternating drives. For example, if your system has four redo log groups, each group with two members, then the following scenario should used to separate disk access:

Four groups with two members each = eight logfiles labeled 1a, 1b, 2a, 2b, 3a, 3b, 4a, and 4b.

This requires at least four disks, plus one disk for archived files.

Example Configurations for Redo Logs and Archive Logs Figure 15–1 illustrates how redo members should be distributed across disks to minimize contention.

Figure 15–1 Distributing Redo Members Across Disks



In the above example, LGWR switched out of log group one (member 1a and 1b) and writes to log group two (2a and 2b). Concurrently, the archiver process reads from group one and writes 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. Hence, a parallel write does not take longer than the longest possible single-disk write.

Because redo logs are written serially, drives dedicated to redo log activity generally require limited head movement. This significantly accelerates log writing.

Three Sample Configurations

Below are three high-level examples of configuring I/O systems. These examples include sample calculations that define the disk topology, stripe sizes, and so on.

Stripe Everything Across Every Disk

The simplest approach is to build one giant volume, striped across all available disks. To account for recoverability, the volume is mirrored (RAID 1). The striping

unit per disk should be larger than the maximum I/O size for the frequent I/O operations. This provides adequate performance for most cases.

Move Archive Logs to Different Disks

If archive logs are striped on the same set of disks as other files, then any I/O requests on those disks when redo logs are being archived could suffer. Moving archive logs to separate disks accomplishes the following:

- Archive at very high rate (sequential I/O).
- Nothing else is affected by the degraded response time on the archive destination disks.

The number of disks for archive logs is determined by the rate of archive log generation and the amount of archive storage required.

Move Redo Logs to Separate Disks

In high-update OLTP systems, the redo logs are write-intensive. Moving the redo log files to disks separate from other disks and from archived redo log files has the following benefits:

- Writing redo logs is performed at the highest possible performance. Hence, transaction processing performance is at its best.
- Writing of the redo logs is not impaired with any other I/O.

The number of disks for redo logs is mostly determined by the redo log size, which is generally small compared to current technology disk sizes. Typically, a configuration with two disks (possibly mirrored to four disks for fault tolerance) is adequate. In particular, by having the redo log files alternating on two disks, writing redo log information to one file does interfere with reading a completed redo log for archiving.

Oracle-Managed Files

For systems where a file system can be used to contain all Oracle data, database administration is simplified by using Oracle-managed files. Oracle internally uses standard file system interfaces to create and delete files as needed for tablespaces, tempfiles, online logs, and controlfiles. Administrators only specify the file system directory to be used for a particular type of file.

Oracle ensures that a unique file is created and then deleted when it is no longer needed. This reduces corruption caused by administrators specifying the wrong file, reduces wasted disk space consumed by obsolete files, and simplifies creation

of test and development databases. It also makes development of portable third party tools easier, because it eliminates the need to put operating system specific file names in SQL scripts.

New files can be created as managed files, while old ones are administered in the old way. Thus, a database can have a mixture of Oracle-managed and manually-managed files.

Note: Oracle-managed files cannot be used with raw devices.

Tuning Oracle-Managed Files

- Because Oracle-managed files require the use of a file system, DBAs give up control over how the data is laid out. Therefore, it is important to correctly configure the file system.
- The Oracle-managed file system should be built on top of a logical volume manager that supports striping. For load balancing and improved throughput, the disks in the Oracle-managed file system should be striped.
- Oracle-managed files work best if used on a logical volume manager that supports dynamically extensible logical volumes. Otherwise, the logical volumes / file system should be configured as large as possible.
- Oracle-managed files work best if the file system provides large extensible files.

See Also: *Oracle9i Database Administrator's Guide* for detailed information on using Oracle-managed files

Choosing Data Block Size

This section lists considerations in choosing database block size for optimal performance.

Reads Regardless of the size of the data, the goal is to minimize the number of reads required to retrieve the desired data.

- If the rows are small and access is predominantly random, then choose a smaller block size.
- If the rows are small and access is predominantly sequential, then choose a larger block size.
- If the rows are small and access is both random and sequential, then it might be effective to choose a larger block size.

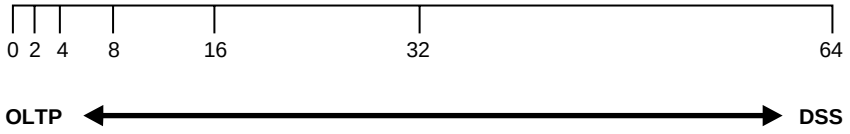
- If the rows are large (for example, LOB data), then choose a larger block size.

Writes For high-concurrency online transaction processing (OLTP) systems, consider appropriate values for `INITRANS`, `MAXTRANS` and `FREELISTS` when using a larger block size, because these parameters affect the degree of update concurrency allowed within a block. However, specifying the value for `FREELISTS` is not necessary when using automatic segment-space management.

If you are uncertain about which block size to choose, then try a database block size of 8K for most large transactional processing systems. This represents a good compromise and is usually effective.

Figure 15–2 illustrates the suitability of various block sizes to OLTP or decision support (DSS) applications.

Figure 15–2 Block Size (in KB) and Application Type



See Also: Your Oracle operating system-specific documentation for information on the minimum and maximum block size on your platform

Block Size Advantages and Disadvantages

Table 15–3 lists the advantages and disadvantages of different block sizes.

Table 15–3 Block Size Advantages and Disadvantages

Block Size	Advantages	Disadvantages
Smaller	Good for small rows with lots of random access. Reduces block contention.	Has relatively large space overhead due to metadata (that is, block header). Not recommended for large rows. There might only be a few rows stored per block, or worse, row chaining if a single row does not fit into a block,
Larger	There is less overhead, so there is more room to store data. It is possible to read a number of rows into the buffer cache with a single I/O (depending on row size and block size). Good for sequential access or very large rows (such as LOB data).	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 waste 7,950 bytes in the buffer cache when doing random access. Large block size is not good for index blocks used in an OLTP environment, because they increase block contention on the index leaf blocks.

16

Understanding Operating System Resources

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
- Solving Operating System Problems
- Understanding CPU
- Finding System CPU Utilization

See Also:

- Your Oracle platform-specific documentation and your operating system vendor's documentation
- *Oracle9i Database Performance Methods* for information on operating system statistics

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 an application experiences excessive buffer busy waits, then the number of system calls increases. If you reduce the buffer busy waits by tuning the application, then the number of system calls decreases. Similarly, if you turn on the Oracle initialization parameter `TIMED_STATISTICS`, then the number of system calls increases. If you turn it off, then system calls decrease.

> **See Also:** Your Oracle platform-specific documentation and your operating system vendor's documentation

Using Hardware and Operating System 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 nondatabase 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.

Evaluating Raw Devices

Evaluate the use of raw devices on the system. Using raw devices can involve a significant amount of work, but can also provide significant performance benefits.

Raw devices impose a penalty on full table scans, but might 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 a 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 might not be possible.

Raw devices on NT are similar to UNIX raw devices; however, all NT devices support write through cache.

> **See Also:** Chapter 15, "I/O Configuration and Design" for a discussion on raw devices versus UNIX file system (UFS)

Using 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 the 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 might exacerbate the effects of contention. The operating system might 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 can 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 can 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 bind to the CPU.

Using 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. Oracle does not certify these system features for compatibility with specific release levels.
- *Oracle9i Database Concepts* and *Oracle9i Database Administrator's Guide* for more information about Oracle Database Resource Manager

Solving Operating System Problems

This section provides hints for tuning various systems by explaining the following topics:

- Performance Hints on UNIX-Based Systems
- Performance Hints on NT Systems
- Performance Hints on Mainframe Computers

Familiarize yourself with platform-specific issues so that you know what performance options the 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: Your Oracle platform-specific documentation and your operating system vendor's documentation

Performance Hints 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. The goal should be to run 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 the following:

- 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 an application can perform.

Performance Hints on NT Systems

On NT systems, as with UNIX-based systems, 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 Hints 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 might 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.

Understanding CPU

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:

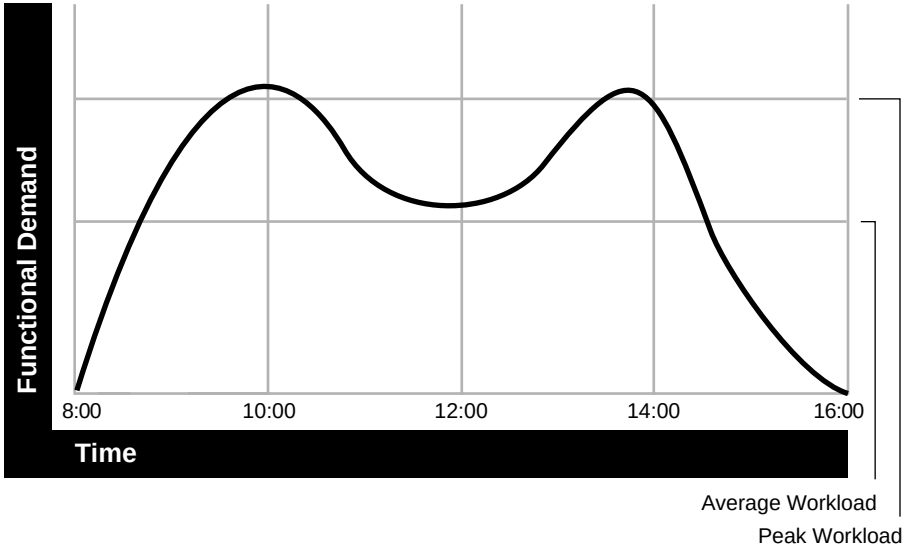
- System is idle (when little Oracle and non-Oracle activity exists)
- System at average workloads
- System at peak workloads

You can capture various workload snapshots using Statspack or the UTLBSTAT/UTLESTAT utility. 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:** Chapter 21, "Using Statspack" for more information on Statspack and UTLBSTAT/UTLESTAT

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 can be acceptable. Even 30% utilization at a time of low workload can be understandable. However, if your system shows high utilization at normal workload, then there is no room for a peak workload. For example, Figure 16–1 illustrates workload over time for an application having peak periods at 10:00 AM and 2:00 PM.

Figure 16–1 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 consumption:

   See Also: "Finding System CPU Utilization"

2. Increasing hardware capacity, including changing the system architecture.

   See Also: *Oracle9i Database Performance Methods* for information about improving your system architecture

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: *Oracle9i Database Concepts* and *Oracle9i Database Administrator's Guide* for more information about Oracle's Database Resource Manager

Finding 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.

Checking 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 can result in the page tables becoming too large. This is not an issue on NT.

Checking 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 might be using only 50% of the time allotted to actually perform work.

Client/Server Round Trips The latency of sending a message can 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:** Chapter 15, "I/O Configuration and Design"

Checking Process Management

Check the following process management issues:

Scheduling and Switching The operating system can 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 might have page table entries for every 4K, and a page table entry might 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 shared 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 might not be wise. In this situation, it becomes inefficient for both the application and the system.

17

Configuring Instance Recovery Performance

This chapter offers guidelines for configuring the time to perform instance recovery.

This chapter contains the following sections:

- Understanding Instance Recovery
- Checkpointing and Cache Recovery
- Reducing Checkpoint Frequency to Optimize Runtime Performance
- Configuring the Duration of Cache Recovery
- Monitoring Cache Recovery
- Tuning Transaction 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.

During normal operation, if an instance is shutdown cleanly (for example, using a `SHUTDOWN IMMEDIATE` statement), rather than terminated abnormally, then the in-memory changes that have not already been written to the datafiles on disk are written to disk as part of the checkpoint performed during shutdown.

However, if a single instance database crashes (or if all instances of an Oracle Real Application Cluster configuration crash), then Oracle performs crash recovery at the next startup. If one or more instances of an Oracle Real Application Cluster configuration crash, then a surviving instance performs instance recovery automatically.

Instance and crash recovery occur in two steps: *cache recovery* then *transaction recovery*.

Cache Recovery (Rolling Forward) During the cache recovery step, Oracle applies all committed and uncommitted changes in the redo log files to the affected data blocks. The work required for cache recovery processing is proportional to the rate of change to the database (update transactions per second) and the time between checkpoints.

Transaction Recovery (Rolling Back) To make the database consistent, the changes that were not committed at the time of the crash must be undone (in other words, rolled back). During the transaction recovery step, Oracle applies the rollback segments to undo the uncommitted changes. The work required to do transaction recovery is proportional to the number and size of uncommitted transactions when the system fault occurred.

Checkpointing and Cache Recovery

Periodically, Oracle records a *checkpoint*. A checkpoint is the highest system change number (SCN) such that all data blocks below or at that SCN are know to be written out to the data files. If a failure occurs, then only the redo records containing changes at SCNs higher than the checkpoint need to be applied during recovery. The duration of cache recovery processing is determined by two factors: the number of data blocks that have changes at SCNs higher than the SCN of the checkpoint, and the number of log blocks that need to be read to find those changes.

How Checkpoints Affect Performance

Frequent checkpointing writes dirty buffers to the datafiles more often than otherwise, and so reduces cache recovery time in the event of an instance failure. If checkpointing is frequent, then applying 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 cache recovery phase of recovery is fairly short.

However, in a high-update system, frequent checkpointing can reduce runtime performance, because checkpointing causes DBW*n* processes to perform writes.

Fast Instance Recovery Trade-offs

To minimize the duration of instance recovery, you must force Oracle to checkpoint often, thus keeping the number of redo log records to be applied during recovery to a minimum. However, frequent checkpointing increases the overhead for normal database operations.

If daily operational efficiency is more important than minimizing recovery time, then decrease the frequency of writes to data files due to checkpoints. This should improve operational efficiency, but also increase instance recovery time.

See Also:

- "Reducing Checkpoint Frequency to Optimize Runtime Performance" for information on how to maximize runtime performance
- "Configuring the Duration of Cache Recovery" for information on how to minimize instance recovery time

Reducing Checkpoint Frequency to Optimize Runtime Performance

To reduce the checkpoint frequency and optimize runtime performance, you can do the following:

- Size your online redo log files according to the amount of redo your system generates. A rough guide is to switch logs at most once every twenty minutes. Small log files can increase checkpoint activity and reduce performance.
- Set the value of the `LOG_CHECKPOINT_INTERVAL` initialization parameter (in multiples of physical block size) to zero. This value eliminates interval checkpoints.

- Set the value of the LOG_CHECKPOINT_TIMEOUT initialization parameter to zero. This value eliminates time-based checkpoints.
- Set the value of FAST_START_MTTR_TARGET (and FAST_START_IO_TARGET) to zero to disable fast-start checkpointing.

  See Also: *Oracle9i Database Concepts* for a complete discussion of checkpoints

Configuring the Duration of Cache Recovery

There are several methods for tuning cache recovery to keep the duration of recovery within user-specified bounds. These include the following:

- Use Fast-Start Checkpointing to Limit Instance Recovery Time
- Set LOG_CHECKPOINT_TIMEOUT to Influence the Number of Redo Logs
- Set LOG_CHECKPOINT_INTERVAL to Influence the Number of Redo Logs
- Use Parallel Recovery to Speed up Redo Application

Use Fast-Start Checkpointing to Limit Instance Recovery Time

The Oracle 9*i* Enterprise Edition offers fast-start fault recovery functionality to control instance recovery. This reduces the time required for cache recovery and makes the recovery bounded and predictable by limiting the number of dirty buffers and the number of redo records generated between the most recent redo record and the last checkpoint.

The foundation of fast-start recovery is the fast-start checkpointing architecture. Instead of the conventional event driven (that is, log switching) checkpointing, which does bulk writes, fast-start checkpointing occurs incrementally. Each DBWn process periodically writes buffers to disk to advance the checkpoint position. The oldest modified blocks are written first to ensure that every write lets the checkpoint advance. Fast-start checkpointing eliminates bulk writes and the resultant I/O spikes that occur with conventional checkpointing.

Administrators specify a target (bounded) time to complete the cache recovery phase of recovery with the FAST_START_MTTR_TARGET initialization parameter, and Oracle automatically varies the incremental checkpoint writes to meet that target. The FAST_START_MTTR_TARGET initialization parameter lets you specify in seconds the expected "mean time to recover" (MTTR), which is the expected amount of time Oracle takes to perform crash/instance recovery for a single instance.

FAST_START_MTTR_TARGET

The FAST_START_MTTR_TARGET initialization parameter simplifies the configuration of recovery time from instance or system failure. This parameter lets you specify the number of seconds crash recovery is expected to take. The FAST_START_MTTR_TARGET is internally converted to a set of parameters that modify the operation of Oracle such that recovery time is as close to this estimate as possible.

Note: You should disable or remove the FAST_START_IO_TARGET, LOG_CHECKPOINT_INTERVAL, and LOG_CHECKPOINT_TIMEOUT parameters when using FAST_START_MTTR_TARGET. Setting these parameters to active values interferes with FAST_START_MTTR_TARGET, resulting in a different than expected value V$INSTANCE_RECOVERY.TARGET_MTTR.

The maximum value for FAST_START_MTTR_TARGET is 3600, or one hour. If you set the value to more than 3600, then Oracle rounds it to 3600. There is no minimum value for FAST_START_MTTR_TARGET. However, this does not mean that you can target the recovery time as low as you want. The time to do a crash recovery is limited by the low limit of the target number of dirty buffers, which is 1000, as well as factors such as how long initialization and file open take.

If you set the value of FAST_START_MTTR_TARGET too low, then the effective mean time to recover (MTTR) target will be the best MTTR target the system can achieve. If you set the value of FAST_START_MTTR_TARGET to such a high value that even in the worst-case recovery would not take that long, then the effective MTTR target will be the estimated MTTR in the worst-case scenario (when the whole buffer cache is dirty). Use the TARGET_MTTR column in V$INSTANCE_RECOVERY to see the effective MTTR.

Note: The TARGET_MTTR column in V$INSTANCE_RECOVERY could be different than FAST_START_MTTR_TARGET if the latter is set too low or too high. Periodically check the MTTR_TARGET column in the V$INSTANCE_RECOVERY view and compare it with the parameter setting. Adjust the parameter setting if it is consistently different from the value in the target.

See Also: "Monitoring Estimated MTTR: Example Scenario" on page 17-10 for more information on setting FAST_START_MTTR_TARGET

Set LOG_CHECKPOINT_TIMEOUT to Influence the Number of Redo Logs

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, or 30 minutes.

Set LOG_CHECKPOINT_INTERVAL to Influence the Number of Redo Logs

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.

See Also:

- "Calculating Performance Overhead" on page 17-11
- Chapter 15, "I/O Configuration and Design" for more information on tuning checkpoints

Use Parallel Recovery to Speed up Redo Application

Use parallel recovery to tune the cache recovery phase of recovery. Parallel recovery uses a division of labor approach to allocate different processes to different data blocks during the cache recovery 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 processes to be read into the buffer cache. Crash, instance, and media recovery of datafiles on different disk drives are good candidates for parallel recovery.

Use the RECOVERY_PARALLELISM initialization parameter to specify the number of concurrent recovery processes for instance or crash recovery. To use parallel processing, the value of RECOVERY_PARALLELISM must be greater than 1 and cannot exceed the value of the PARALLEL_MAX_SERVERS initialization parameter.

Note: The RECOVERY_PARALLELISM initialization parameter specifies the number of concurrent recovery processes for instance or crash recovery *only*.

Media recovery is not affected by this parameter. Use the PARALLEL clause in the RECOVER DATABASE statement for media recovery.

Recovery is usually I/O bound on reads to data blocks. Consequently, parallelism at the block level can only help recovery performance if it speeds up total I/Os. Performance on systems with efficient asynchronous I/O typically does not improve significantly with parallel recovery.

Initialization Parameters that Influence Cache Recovery Time

The following initialization parameters influence cache recovery time.

Table 17–1 Initialization Parameters Influencing Cache Recovery

Parameter	Purpose
FAST_START_MTTR_TARGET	Lets you specify in seconds the expected "mean time to recover" (MTTR), which is the expected amount of time Oracle takes to perform recovery and startup the instance.
FAST_START_IO_TARGET	This parameter has been deprecated in favour of FAST_START_MTTR_TARGET. This parameter specifies the upper limit on the number of dirty buffers.
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 blocks generated between the most recent redo record and the checkpoint.
RECOVERY_PARALLELISM	Specifies the number of concurrent recovery processes to be used in instance or crash recovery.

Note: Oracle recommends using the FAST_START_MTTR_TARGET parameter to control the duration of startup after instance failure. Fast-start checkpointing is only available with Enterprise Edition.

The FAST_START_IO_TARGET parameter has been deprecated in favor of the FAST_START_MTTR_TARGET parameter.

The parameter DB_BLOCK_MAX_DIRTY_TARGET has been removed.

Monitoring Cache Recovery

Use the V$INSTANCE_RECOVERY view to see the current recovery parameter settings. You can also use statistics from this view to calculate which parameter has the greatest influence on checkpointing. V$INSTANCE_RECOVERY contains the columns shown in Table 17–2.

Note: The last three fields in V$INSTANCE_RECOVERY are new with Oracle9*i*, and they are the most important. With the initialization parameter FAST_START_MTTR_TARGET, the other seven fields of V$INSTANCE_RECOVERY are less useful.

Table 17–2 V$INSTANCE_RECOVERY View

Column	Description
RECOVERY_ESTIMATED_IOS	Contains the number of dirty buffers in the buffer cache. (In Standard Edition, the value of this field is always NULL).
ACTUAL_REDO_BLKS	Current number of redo blocks required to be read 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 three columns (LOG_FILE_SIZE_REDO_BLKS, LOG_CHKPT_TIMEOUT_REDO_BLKS, LOG_CHKPT_INTERVAL_REDO_BLKS).
LOG_FILE_SIZE_REDO_BLKS	Number of redo blocks to be processed during recovery corresponding to 90% of the size 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.
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	This field is obsolete. It is retained for backward compatibility. The value of this field is always NULL.
TARGET_MTTR	Effective mean time to recover (MTTR) target in seconds. Usually, it should be equal to the value of the FAST_START_MTTR_TARGET parameter. If FAST_START_MTTR_TARGET is set to such a small value that it is impossible to do a recovery within its time frame, then the TARGET_MTTR field contains the effective MTTR target, which is larger than FAST_START_MTTR_TARGET. If FAST_START_MTTR_TARGET is set to such a high value that even in the worst-case (the whole buffer cache is dirty) recovery would not take that long, then the TARGET_MTTR field contains the estimated MTTR in the worst-case scenario. This field is 0 if FAST_START_MTTR_TARGET is not specified.
ESTIMATED_MTTR	The current estimated mean time to recover (MTTR) in the number of seconds based on the number of dirty buffers and log blocks (gives the current estimated MTTR even if FAST_START_MTTR_TARGET is not specified).
CKPT_BLOCK_WRITES	Number of blocks written by checkpoint writes.

See Also: *Oracle9i Database Reference* for more information on the V$INSTANCE_RECOVERY view

Monitoring Estimated MTTR: Example Scenario

The TARGET_MTTR field of V$INSTANCE_RECOVERY contains the MTTR target in effect. The ESTIMATED_MTTR field of V$INSTANCE_RECOVERY contains the estimated MTTR should a crash happen right away. Query these two fields to see if the system can keep up with your specified MTTR target.

For example, assume the initialization parameter setting is as follows:

```
FAST_START_MTTR_TARGET = 6      # seconds
```

Execute the following query after database open:

```
SELECT TARGET_MTTR, ESTIMATED_MTTR, CKPT_BLOCK_WRITES
FROM V$INSTANCE_RECOVERY;
```

Oracle responds with the following:

```
TARGET_MTTR ESTIMATED_MTTR CKPT_BLOCK_WRITES
18          15             0
```

You see that TARGET_MTTR is 18 seconds, which is higher than the value of FAST_START_MTTR_TARGET specified (6 seconds). This means that it is impossible to recover the database within 6 seconds. 18 seconds is the minimum MTTR target that the system can achieve.

The 18 second minimum is calculated based on the absolute low limit of 1000 blocks on the target of number of dirty buffers (The deprecated initialization parameter FAST_START_IO_TARGET follows this low limit; that is, you cannot set FAST_START_IO_TARGET below 1000). The ESTIMATED_MTTR field contains the estimated number of log blocks generated since the last checkpoint. Because the database has just opened, the system contains few dirty buffers. That is why ESTIMATED_MTTR can be lower than the minimum possible TARGET_MTTR.

Now assume that you use the following statement to modify FAST_START_MTTR_TARGET:

```
ALTER SYSTEM SET FAST_START_MTTR_TARGET = 30;
```

Reissue the query to V$INSTANCE_RECOVERY, and Oracle responds with the following:

```
TARGET_MTTR ESTIMATED_MTTR CKPT_BLOCK_WRITES
30          15             0
```

The ESTIMATED_MTTR field is still 15 seconds, which means that the estimated MTTR at the current time (should a crash happen immediately) is still 15 seconds. This is because no new redo is written, and no data block has become dirty.

Assume that you make tremendous updates to the database and query V$INSTANCE_RECOVERY immediately afterwards. Oracle responds with the following:

```
TARGET_MTTR ESTIMATED_MTTR CKPT_BLOCK_WRITES
30          36             54367
```

You see that the effective MTTR target is 30 seconds. The estimated MTTR at the current time (should a crash happen immediately) is 36 seconds. This is fine. This means that some checkpoints writes might not have finished yet, so the buffer cache contains more dirty buffers than targeted.

Assume that you wait for one minute and reissue the query to V$INSTANCE_RECOVERY. Oracle responds with the following:

```
TARGET_MTTR ESTIMATED_MTTR CKPT_BLOCK_WRITES
30          31             55230
```

The estimated MTTR at this time has dropped to 31 seconds. This is because more dirty buffers have been written out during this period. This is shown by the increase of CKPT_BLOCK_WRITES field of V$INSTANCE_RECOVERY.

Note: The number of physical writes minus the number of physical writes non checkpoint (from V$SYSSTAT) equals the field CKPT_BLOCK_WRITES in V$INSTANCE_RECOVERY.

Calculating Performance Overhead

To calculate performance overhead, use the V$SYSSTAT view. For example, assume that you execute the following query:

```
SELECT NAME, VALUE
FROM V$SYSSTAT
WHERE NAME IN ('physical reads','physical writes',
               'physical writes non checkpoint');
```

Oracle responds with the following:

```
NAME                                 VALUE
physical reads                        2376
physical writes                      14932
physical writes non checkpoint       11165
```

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 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_MTTR_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))] x 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 of FAST_START_MTTR_TARGET.

The number of extra writes caused by setting FAST_START_MTTR_TARGET to a nonzero value is application-dependent; it is not dependent on cache size.

Calculating Performance Overhead: Example Scenario

As an example, assume the initialization parameter setting is as follows:

```
FAST_START_MTTR_TARGET = 90  # 90 seconds
```

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 the following:

```
Name                              Value
physical reads                     2376
physical writes                   14932
physical writes non checkpoint    11165
```

The physical write checkpoint statistics can also be found in the CKPT_BLOCK_WRITES field of the V$INSTANCE_RECOVERY view. For example:

```
SELECT CKPT_BLOCK_WRITES
FROM V$INSTANCE_RECOVERY;
```

Oracle responds with the following:

```
CKPT_BLOCK_WRITES    3767
```

It is consistent with the result from V$SYSSTAT: 3767 = 14932 - 11165.

After making updates for a few hours, you reissue the query. Oracle responds with the following:

```
Name                              Value
physical reads                     3011
physical writes                   17467
physical writes non checkpoint    13231
```

Substitute the values from the SELECT statements in the formula described above 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 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_MTTR_TARGET to 60. 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 the following:

```
Name                                Value
physical reads                       4652
physical writes                     28864
physical writes non checkpoint      21784
```

After making updates, reissue the query. Oracle responds with the following:

```
Name                                Value
physical reads                       6000
physical writes                     35394
physical writes non checkpoint      26438
```

Calculate how much performance overhead you are incurring using the values from the 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.

Calibrating the MTTR

The FAST_START_MTTR_TARGET initialization parameter calculates internal system trigger values to limit the length of the redo log and the number of dirty data buffers in the data cache. This calculation uses estimated times to read a redo block and to read and write a data block.

Initially, internal defaults are used. These defaults are replaced by execution time estimates during system operation. However, the best values are obtained from measurements taken from an actual recovery from a failure. To effectively align FAST_START_MTTR_TARGET, perform several instance recoveries to ensure that the time to read a redo block and the time to read and write a data block are recorded accurately.

Before doing instance recoveries to calibrate the FAST_START_MTTR_TARGET, decide whether FAST_START_MTTR_TARGET is being calibrated for a database crash or a hardware crash. This is a consideration if your database files are stored in a file system or if your I/O subsystem has a memory cache, because there is a considerable difference in the read and write time to disk depending on whether or not the files are cached. The workload being run during the instance recovery should be a very good representation of the average workload on the system to ensure that the amount of redo records generated are similar.

Tuning Transaction Recovery

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 Oracle9*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 cache recovery phase of recovery completes. If a user attempts 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 transaction recovery what parallel recovery is to cache recovery.

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. SMON automatically decides when to begin parallel rollback and disperses the work among several parallel processes: process one rolls back one transaction, process two rolls back a second transaction, and so on.

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 eight transactions require recovery with one parallel process assigned to each transaction. The transactions are all similar in size except for transaction five, 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 five among the processes: process one takes one part, process two takes another part, and so on.

You control the number of processes involved in transaction recovery by setting the initialization 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 cannot exceed twice the value of the CPU_COUNT initialization parameter.
HIGH	Specifies that the number of recovery servers cannot exceed four times the value of the CPU_COUNT initialization parameter.

Parallel Rollback in an Oracle Real Application Clusters Configuration In Oracle Real Application Clusters, 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 views. 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:

- *Oracle9i Database Administrator's Guide* and *Oracle9i Real Application Clusters Concepts* for more information on managing undo space
- *Oracle9i Real Application Clusters Deployment and Performance* for more information on fast-start parallel rollback in an Oracle Real Application Clusters environment
- *Oracle9i Database Reference* for more information about initialization parameters
- *Oracle Net Services Administrator's Guide* for information on transparent application failure (TAF)

18

Configuring Undo and Temporary Segments

This chapter contains the following topics:

- Configuring Undo Segments
- Configuring Temporary Tablespaces

Configuring Undo Segments

Oracle provides *automatic undo management*, which completely automates the management of undo data. A database running in automatic undo management mode transparently creates and manages undo segments. Oracle Corporation strongly recommends using automatic undo management, because it significantly simplifies database management and removes the need for any manual tuning of undo (rollback) segments. Manual undo management using rollback segments is supported for backward compatibility reasons.

Configuring Automatic Undo Management

To configure automatic undo, you simply create an undo tablespace, and determine the maximum retention time for undo data kept in that tablespace.

> **See Also:** *Oracle9i Database Administrator's Guide* for detailed information on how to configure automatic undo

Configuring Rollback Segments

Automatic undo management is the preferred way of handling rollback space. Automatic undo management lets you allocate undo space in a single undo tablespace, instead of distributing undo space in a set of statically allocated rollback segments. The creation and allocation of space amongst the undo segments is handled automatically by the Oracle server.

With rollback segments, one or more tablespaces are created; rollback segments are manually created in those tablespaces. The number and size of the rollback segments must be determined by the DBA.

Determining the Number and Size of 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, avoid dynamic space management in rollback segments.

Table 18–1 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 18–1 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

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:** *Oracle9i Database Administrator's Guide* for information on choosing values for this parameter, monitoring rollback segment shrinking, and adjusting the OPTIMAL parameter

For Long Queries Assign large rollback segments to transactions that modify data that is concurrently selected by long queries. Such queries might 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; avoid it 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 two extents, each approximately 10 kilobytes in size. To best avoid contention, create many rollback segments and assign each transaction to its own rollback segment.

Configuring Temporary Tablespaces

Configuring the temporary tablespace helps optimize disk sort performance. This involves choosing good storage clauses and the correct type of tablespace to use for sorting.

Choosing the default storage clause for the sort tablespace includes the following:

- Setting `PCTINCREASE` to zero
- Setting `INITIAL` and `NEXT` to the same size and a factor of `SORT_AREA_SIZE`

Choosing the correct type of tablespace makes disk sorting more efficient. The various tablespaces that could be used for disk sorting include the following:

- Temporary Tablespaces
- Tablespaces of Type TEMPORARY
- Permanent Tablespaces

Temporary Tablespaces These are the most efficient tablespaces for disk sorts. Characteristics of a temporary tablespace include the following:

- Space management (extent allocation and deallocation) is locally managed. Therefore, use of the ST-enqueue is avoided.
- The sort segment created for each instance is reused (it is only dropped if the tablespace is dropped). With Oracle Real Application Clusters, one sort segment is required per instance.
- All processes performing sorts reuse existing sort extents of the sort segment, rather than allocating a segment (and potentially many extents) for each sort. If an insufficient number of extents exist for the number of sorts currently in

operation, then the required extents are added once per instance startup. They are recycled thereafter.

- Temporary tablespaces can be striped as described in "Basic I/O Configuration" on page 15-65. Only when the tablespaces is creating an I/O bottleneck should the tablespace be segregated from others.
- To create temporary tablespaces, use the CREATE TEMPORARY TABLESPACE statement.

  See Also:

  - *Oracle9i Database Concepts* for more information on temporary tablespaces
  - *Oracle9i SQL Reference* for more information on using the CREATE TEMPORARY TABLESPACE statement

Tablespaces of Type TEMPORARY After temporary tablespaces, these are next best tablespaces to use for sort operations. Characteristics of tablespaces of type TEMPORARY include the following:

- Space management is dictionary-managed. Therefore, they use the ST-enqueue.
- The sort segment for an instance is dropped on instance startup and re-created when the first sort is performed. Extents are then allocated as needed.
- Although space management is dictionary-managed, the frequency of space allocation and deallocation is reduced. All processes performing sorts reuse the existing sort extents of a single sort segment, rather than allocating an deallocating extents and segments for each sort. With Oracle Real Application Clusters, one sort segment is required per instance. If insufficient extents exist for the number of sorts, then the required extents are added once per instance startup, and are recycled thereafter. This in turn reduces the load on the ST enqueue considerably.
- To create tablespaces of type TEMPORARY, use the CREATE TABLESPACE or ALTER TABLESPACE statements with the TEMPORARY clause.

  See Also: *Oracle9i SQL Reference* for more information on using the TEMPORARY clause

Permanent Tablespaces Permanent tablespaces (which are not of type TEMPORARY) are least efficient for performance of disk sorts. This is because of the following reasons:

- The ST-enqueue is used for allocation and de-allocation of each extent allocated to a sort segment.
- Sort-segments are not reused. Each process performing a disk sort creates then drops it's own sort segment. In addition, a single sort operation can require the allocation and deallocation of many extents, and each extent allocation requires the ST-enqueue.

For optimal performance, the best choice is to use temporary tablespaces. They bypass the need of using the ST enqueue for space management and have the additional benefit of reuse of sort extents.

Table 18–2 Sort Tablespaces

Tablespace Type	Space Management	Sort Segment Reuse	Sort Extent Reuse	Creation Statement
Temporary (uses tempfiles)	Locally Managed	Yes. Sort segment is not dropped	Always	CREATE TEMPORARY TABLESPACE
Type TEMPORARY (uses datafiles)	Dictionary Managed	Sort segment is recreated on instance startup	Yes. Extents are reused until sort segment is dropped on instance startup	CREATE TABLESPACE ... TYPE TEMPORARY
Permanent	Dictionary Managed	No. Sort segments are not reused	No. Sort segments are not reused	CREATE TABLESPACE

Temporary tablespaces, and tablespaces of type TEMPORARY, cannot contain permanent objects, such as tables or rollback segments.

> **See Also:** *Oracle9i SQL Reference* for more information about the syntax of the CREATE TABLESPACE and ALTER TABLESPACE statements

19

Configuring Shared Servers

This chapter contains the following topics:

- Configuring the Number of Shared Servers

Configuring the Number of Shared Servers

Performance of certain database features can improve when a shared server architecture is used, and performance of certain database features can degrade slightly when a shared server architecture is used. For example, with BFILEs, if a query has not completed, then BFILEs must close and reopen in other servers when migrated. With parallel execution, a session can be prevented from migrating to another shared server while they are active.

A session can remain nonmigratable after a request from the client has been processed. Use of the mentioned features can make sessions nonmigratable, 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 nonmigratable among shared servers.

When using these features, you might need to configure more shared servers. This is because some servers might be bound to sessions for an excessive amount of time.

This section discusses how to reduce contention for processes used by Oracle's 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:

- *Oracle9i Database Reference* for detailed information about these views
- *Oracle Enterprise Manager Concepts Guide* for information about how to use the Tuning Pack to monitor these statistics

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 the prefix MAX_ 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 shared server 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 might need to add more dispatchers. A general rule is to examine V$DISPATCHER_RATE statistics during both light and heavy system use periods. After identifying your shared server 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 add dispatcher processes and how to enable connection pooling.

Adding Dispatcher Processes

Add dispatcher processes while Oracle is running with the SET option of the ALTER SYSTEM statement to increase the value for the DISPATCHERS initialization parameter.

The total number of dispatcher processes is limited by the value of the initialization parameter MAX_DISPATCHERS. You might need to increase this value before adding dispatcher processes. The default value of this parameter is five, and the maximum value varies depending on your operating system.

> **See Also:** *Oracle9i Database Administrator's Guide* and *Oracle Net Services Administrator's Guide* for more information on adding dispatcher processes

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.

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:

```
DISPATCHERS = "(PROTOCOL=TCP)(POOL=ON)(TICK=1)"
```

The optional attribute POOL enables the Oracle Net connection pooling feature. TICK is the size of a network TICK in seconds. The TICK default is 15 seconds.

> **See Also:** *Oracle9i SQL Reference* and the *Oracle Net Services Administrator's Guide* for more information about the DISPATCHERS parameter and its options

Enabling Session Multiplexing

Multiplexing is used by a connection manager process to establish and maintain connections from multiple users to individual dispatchers. For example, several user processes can connect to one dispatcher by way of a single connection from a connection manager process.

The connection manager manages communication from users to the dispatcher by way of a shared connection. At any one time, zero, one, or a few users might need the connection, while other user processes linked to the dispatcher by way of the

connection manager process are idle. This way, multiplexing is beneficial because 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:

```
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. The following 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 show 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 the following query:

```
SELECT COUNT(*) "Shared Server Processes"
    FROM V$SHARED_SERVER
    WHERE STATUS != 'QUIT';
```

The result of this query could look like the following:

```
SHARED SERVER PROCESSES
----------------------
10
```

If you detect resource contention with shared servers, 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 might want to create more resources to reduce shared server process contention. Do this by modifying the optional server process parameters, as explained below.

Setting and Modifying Shared Server Processes

This section explains how to set optional parameters affecting processes for the shared server architecture. This section also explains how and when to modify these parameters to tune performance.

The following static initialization parameters are discussed in this section:

- MAX_DISPATCHERS
- MAX_SHARED_SERVERS

This section also describes the following initialization / session parameters:

- DISPATCHERS
- SHARED_SERVERS

Values for the initialization parameters MAX_DISPATCHERS and MAX_SHARED_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 MAX_DISPATCHERS sets the limit on the number of dispatchers for all DISPATCHERS' dispatcher values.

You can also define starting values for the number of dispatchers and servers by setting the DISPATCHERS parameter's DISPATCHER attribute and the SHARED_SERVERS parameter. After system startup, you can dynamically reset 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 MAX_SHARED_SERVERS is dependent on the value of SHARED_SERVERS. If SHARED_SERVERS is less than or equal to 10, then MAX_SHARED_SERVERS defaults to 20. If SHARED_SERVERS is greater than 10, then MAX_SHARED_SERVERS defaults to two times the value of SHARED_SERVERS.

Self-adjusting Shared Server Architecture Features

When the database starts, SHARED_SERVERS is the number of shared servers created. Oracle does 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 MAX_SHARED_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 might 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 MAX_SHARED_SERVERS and the average wait time in the request queue is still unacceptable, then you might improve performance by increasing the MAX_SHARED_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 SHARED_SERVERS. You can change the value of this parameter in the initialization parameter file, or alter it using the SHARED_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 Shared Server Highwater Mark Equal to MAX_SHARED_SERVERS

This is the first stage in troubleshooting a shared server architecture. 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 MAX_SHARED_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$SHARED_SERVER_MONITOR;
```

The output is:

```
MAXIMUM_CONNECTIONS SERVERS_STARTED SERVERS_TERMINATED SERVERS_HIGHWATER
------------------- --------------- ------------------ ----------------
                 60              30                 30               50
```

If the system is not performing well, then HIGHWATER should not be equal to the parameter MAX_SHARED_SERVERS regularly or for a long period of time.

> **See Also:** *Oracle9i Database Reference* for complete information on V$SHARED_SERVER_MONITOR

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 MAX_SHARED_SERVERS parameter gradually until you begin to swap. If swapping occurs due to the shared server, then reduce 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 MAX_SHARED_SERVERS is through trial and error.

To change the MAX_SHARED_SERVERS, first edit the initialization parameter file. Save the file and restart the instance. Remember that setting SHARED_SERVERS to MAX_SHARED_SERVERS should only be done if you are sure that you will be using the machine at 100% all the time. Keep in mind the following rules:

- SHARED_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.
- MAX_SHARED_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.

Determining the Optimal Number of Dispatchers and Shared Servers

As mentioned, SHARED_SERVERS determines the number of shared servers activated at instance startup. The default setting for SERVER_SERVERS is one, which is the default setting when 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 MAX_DISPATCHERS and 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 Shared Server Use with Concurrent Shared Server Use

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 SHARED_SERVERS to zero. This temporarily disables additional use of shared servers. Resetting SHARED_SERVERS to a positive value enables shared servers for all current users.

See Also:

- *Oracle9i Database Reference* for information about dispatchers, (see the description of the V$DISPATCHER and V$DISPATCHER_RATE views)
- *Oracle9i SQL Reference* for more information about the ALTER SYSTEM statement
- *Oracle9i Database Administrator's Guide* for more information on changing the number of shared servers

Part IV

System-Related Performance Tools

Part IV provides information about Oracle's system-related performance tools.

The chapters in this part are:

- Chapter 20, "Oracle Tools to Gather Database Statistics"
- Chapter 21, "Using Statspack"

20

Oracle Tools to Gather Database Statistics

Effective data collection and analysis is essential for identifying and correcting system performance problems. Oracle provides a number of tools that allow a performance engineer to gather information regarding instance and database performance. This chapter explains why performance data gathering is important, and it describes how to use available tools.

This chapter contains the following sections:

- Overview of Tools
- Principles of Data Gathering
- Interpreting Statistics
- Oracle Enterprise Manager Diagnostics Pack
- Statspack
- V$ Performance Views

Overview of Tools

Oracle Enterprise Manager Diagnostics Pack, with a graphical user interface, is the most feature-rich performance tool. It provides data analysis and collection of operating system statistics.

Statspack and **BSTAT/ESTAT** are command-line interface tools that gather instance related performance data. Statspack is the successor of BSTAT/ESTAT, with significantly increased functionality over the BSTAT/ESTAT tool.

V$ views can be queried using SQL. They contain dynamic performance data related to an Oracle instance performance. The data is lost when the instance is shut down.

Principles of Data Gathering

To effectively diagnose a performance problem, it is vital to have an established performance baseline for later comparison when the system is running poorly. Without a baseline data point, it can be very difficult to identify new problems. For example, perhaps the volume of transactions on the system has increased, or the transaction profile or application has changed, or the number of users has increased.

Although Oracle Enterprise Manager, Statspack and BSTAT/ESTAT, and the V$ views have different interfaces (GUI, command line, SQL), the majority of data they collect and report on is extracted from the V$ views. Because the V$ views are based on memory-resident data, when an instance is shut down, the data related to the instance is lost.

In order to perform analysis of data from one day to the next, data visible through the V$ views must be saved. On each instance startup, the memory resident V$ views are reinitialized; hence, to determine what has changed within any particular period, you must calculate the difference in the performance data. This is done by subtracting the statistic values at the beginning of the period from the statistic values at the end of the period. This gives you the activity of the instance during that period, or the *delta*.The delta of each statistic can then be normalized (for example, per second or per transaction).

The delta of *all* statistics for a period of time can be considered a baseline, assuming the response time and operations performed on the instance were representative of some typical load on your system (that is, batch, online, or both).

Each of the tools Oracle provides (except for the V$ views themselves) has a mechanism for saving this data and for determining the delta.

It is also important to gather operating system and network statistics. These can then be correlated with the Oracle performance data. If you are using Statspack, BSTAT/ESTAT, or your own tool, you should devise a mechanism for collecting operating system statistics. With Oracle Enterprise Manager, this capability is built-in.

Interpreting Statistics

When initially examining performance data, you can formulate potential theories by examining your statistics. One way to ensure that your interpretation of the statistics is correct is to perform cross-checks with other data. This establishes whether a statistic or event is really of interest.

Some pitfalls are discussed below:

- Hit Ratios

  When tuning, it is common to compute a ratio that helps determine whether there is a problem. Such ratios include the buffer cache hit ratio, the soft-parse ratio, and the latch hit ratio. These ratios should not be used as 'hard and fast' identifiers of whether there is or is not a performance bottleneck. Rather, they should be used as indicators. In order to identify whether there is a bottleneck, other related evidence should be examined.

- Wait Events with Timed Statistics

  Setting TIMED_STATISTICS to true at the instance level directs the Oracle server to gather wait time for events, in addition to wait counts already available. This data is useful for comparing the total wait time for an event to the total elapsed time between the performance data collections. For example, if the wait event accounts for only 30 seconds out of a two hour period, then there is probably little to be gained by investigating this event, even though it may be the highest ranked wait event when ordered by time waited. However, if the event accounts for 30 minutes of a 45 minute period, then the event is worth investigating.

- Comparing Oracle Statistics with Other Factors

  When looking at statistics, it is important to consider other factors that influence whether the statistic is of value. Such factors include the user load and the hardware capability. Even an event that had a wait of 30 minutes in a 45 minute snapshot might not be indicative of a problem if you discover that there were 2000 users on the system, and the host hardware was a 64 node machine.

- Wait Events without Timed Statistics

  If `TIMED_STATISTICS` is false, then the amount of time waited for an event is not available. Therefore, it is only possible to order wait events by the number of times each event was waited for. Although the events with the largest number of waits might indicate the potential bottleneck, they might not be the main bottleneck. This can happen when an event is waited for a large number of times, but the total time waited for that event is small. The converse is also true: an event with fewer waits might be a problem if the wait time is a significant proportion of the total wait time. Without having the wait times to use for comparison, it is difficult to determine whether a wait event is really of interest.

- Idle Wait Events

  Oracle uses some wait events to indicate if the Oracle server process is idle. Typically, these events are of no value when investigating performance problems, and they should be ignored when examining the wait events.

- Computed Statistics

  When interpreting computed statistics (such as per second rates, per transaction rates, or ratios), it is important to cross-verify the computed statistic with the actual statistic counts. This confirms whether the derived rates are really of interest: small statistic counts usually can discount an unusual ratio. For example, on initial examination, a soft-parse ratio of 50% generally indicates a potential tuning area. If, however, there was only one hard parse and one soft parse during the data collection interval, then the soft-parse ratio would be 50%, even though the statistic counts show this is not an area of concern. In this case, the ratio is not of interest due to the low raw statistic counts.

Oracle Enterprise Manager Diagnostics Pack

Oracle Enterprise Manager (EM) Diagnostics Pack captures related operating system, middle-tier, and application performance data, in addition to instance performance data, allowing end-to-end diagnostics.

The Diagnostics Pack can automatically analyze this performance data, display it in a graphical interface, and use alerts to immediately direct you to any performance problems. You can be alerted automatically via email or page when a problem is detected. Oracle Enterprise Manager also includes an integrated diagnostics methodology that uses guided drilldowns and expert advice to help you quickly resolve performance issues.

EM also lets you store the captured data in a separate performance repository database. You can store the performance data for multiple databases in the same repository.

The EM Intelligent Agent data gathering service collects performance data on a scheduled basis. A single agent can manage the data collections for all Oracle databases and the operating system of the target node. The data is automatically stored in an historical data repository for performance reporting. Data stored in the repository can be used to analyze many facets of database performance, such as database load, cache allocations and efficiency, resource contention, and high-impact SQL.

Performance data collections can be initiated directly from the EM Console or through the EM Diagnostics Pack - Capacity Planner application. HTML reports of historical performance data can be generated from the EM Console. These reports provide a comprehensive analysis of database system usage and performance, which can be easily accessed and navigated from a browser. EM also provides a graphical real-time Performance Overview for monitoring a subset of these performance metrics using line charts, bar graphs, and so forth.

The Performance Overview charts let you troubleshoot existing performance problems by drilling into performance data to track down the source of a performance bottleneck. For example, a decline in the memory sort percentage can be immediately investigated by drilling down to the sessions and corresponding SQL responsible for high volume sort activity. High-impact SQL statements discovered through this process can be further investigated by launching SQL diagnostic tools in the context of the problem.

See Also: *Oracle Enterprise Manager Concepts Guide*

Statspack

Statspack fundamentally differs from the well known UTLBSTAT / UTLESTAT tuning scripts by collecting more information, and by storing the performance statistics data permanently in Oracle tables, which can later be used for reporting and analysis. The data collected can be analyzed using the report provided, which includes an "instance health and load" summary page, high resource SQL statements, as well as the traditional wait events and initialization parameters.

See Also: Chapter 21, "Using Statspack"

V$ Performance Views

The V$ views are the performance information sources used by all Oracle performance tuning tools. The V$ views are based on memory structures initialized at instance startup. The memory structures (and hence the views that represent them) are maintained throughout the life of the instance automatically by Oracle.

See Also: Chapter 24, "Dynamic Performance Views for Tuning"

If you choose *not* to use an Oracle tool to gather your performance data, then you need to develop your own. You need to save data from the required performance views to an on-disk structure, so that the data can be analyzed and compared with other data collected. Because there are multiple collections, you need a key to identify each collection. This method is very similar to the method used by Statspack.

Below is an example of how to save performance data from a single V$ view to an Oracle table. To implement your own collection tool, you must perform a similar collection mechanism for all essential V$ views.

Note: Oracle recommends using Oracle Enterprise Manager Diagnostics Pack or Statspack to gather performance data. These tools have been designed to capture all of the data needed for performance analysis.

Example - Saving File I/O Data

The following example creates a table that stores the collection data. The table has all V$FILESTAT columns and also includes the collection ID column and the collection date column. A sample SQL statement that reports the delta in this table for the first two collections has also been included.

Create the table, and insert the first data collection:

```
CREATE TABLE coll_filestat AS
  SELECT 1       coll_id       -- collection number
       , sysdate coll_date     -- collection date
       , fs.*
   FROM V$FILESTAT fs;

ALTER TABLE coll_filestat add
   (constraint coll_filestat primary key (coll_id, file#));
```

At the end of the interval, insert second collection:

```
INSERT INTO coll_filestat
SELECT 2                        -- collection number
      , sysdate                 -- collection date
      , fs.*
 FROM V$FILESTAT fs;
```

> **Note:** In order to insert any other collections, you need to keep the collection ID key unique. This could be done using a sequence number. (The sequence number should have value for all V$ views captured in one collection.)

To query for high I/O tablespaces:

```
SELECT t.tablespace_name
      ,   SUM(fs2.phyrds-fs1.phyrds)
        / MAX(86400*(fs2.coll_date-fs1.coll_date)) "Rd/sec"
      ,   SUM(fs2.phyblkrd-fs1.phyblkrd)
        / MAX(86400*(fs2.coll_date-fs1.coll_date)) "Blk/sec"
      ,   SUM(fs2.phywrts-fs1.phywrts)
        / MAX(86400*(fs2.coll_date-fs1.coll_date)) "Wr/sec"
      ,   SUM(fs2.phyblkwrt-fs1.phyblkwrt)
        / MAX(86400*(fs2.coll_date-fs1.coll_date)) "Blk/sec"
  FROM coll_filestat fs1, coll_filestat fs2, dba_data_files t
 WHERE fs2.file#   = fs1.file#
   AND fs2.coll_id = fs1.coll_id + 1
   AND t.file_id   = fs2.file#
 GROUP BY t.tablespace_name
 ORDER BY sum(fs2.phyblkrd+fs2.phyblkwrt-fs1.phyblkrd-fs1.phyblkwrt) DESC;
```

Below is example output from the output of the above select statement:

```
TABLESPACE_N Rd/sec Blk/sec Wr/sec Blk/sec
------------ ------ ------- ------ -------
AP_T_02       287.1  2245.7     .0      .0
PO_T_01       313.5   650.6     .2      .2
RECEIVABLE_T  401.0   613.8    2.4     2.4
INV_T_01      154.3   155.3     .0      .0
APPLSYS_T_01   63.3   139.6     .4      .4
PA_T_01       102.3   102.3     .0      .0
SO_I_01        63.4    63.4   34.5    34.5
TEMP            2.3    45.0    1.9    47.0
RECEIVABLE_I   73.0    73.0     .1      .1
AP_T_03        69.3    69.3     .0      .0
RECEIVABLE_I   65.1    65.1    1.9     1.9
SO_T_01        54.0    57.8    2.9     2.9
SYSTEM         45.2    59.0     .3      .3
PER_T_01       48.0    58.7     .0      .0
AP_T_01        12.9    51.0     .2      .2
SO_T_03        43.0    43.0    1.2     1.2
PER_I_01       30.8    30.8     .0      .0
FA_T_01        22.3    22.3     .0      .0
INV_I_01       20.7    20.7     .7      .7
PO_I_01        19.5    19.5     .7      .7
GSR_T_01       19.2    19.2     .4      .4
INV_I_03       18.3    18.3     .0      .0
ROLL_01         1.4     1.4   14.7    14.7
PA_I_01        14.3    14.3     .2      .2
```

21

Using Statspack

This chapter explains how to install and configure Statspack.

This chapter contains the following section:

- Statspack vs. BSTAT / ESTAT
- How Statspack Works
- Configuring Statspack
- Installing Statspack
- Using Statspack
- Removing Statspack
- Statspack Supplied Scripts
- Statspack Limitations

Statspack vs. BSTAT/ESTAT

Statspack differs from the existing UTLBSTAT/UTLESTAT performance scripts in the following ways:

- Statspack collects more data, including high-resource SQL.
- Statspack precalculates many ratios useful when performance tuning, such as cache hit ratios and per transaction and per second statistics. (Many of these ratios must be calculated manually when using BSTAT/ESTAT).
- Permanent tables owned by PERFSTAT store performance statistics. Instead of creating/dropping tables each time, data is inserted into the preexisting tables. This makes historical data comparisons easier.
- Statspack separates the data collection from the report generation. Data is collected when a snapshot is taken; viewing the data collected is in the hands of the performance engineer when running the performance report.

> **Note:** The term *snapshot* denotes a set of performance statistics gathered at a single time, identified by a unique ID, which includes the snapshot number (or SNAP_ID). This term should not be confused with Oracle's snapshot replication technology.

- Data collection is easy to automate using either DBMS_JOB or an operating system utility.
- Statspack considers a transaction to finish either with a COMMIT or a ROLLBACK, and so calculates the number of transactions as 'user commits' + 'user rollbacks.' BSTAT/ESTAT considers a transaction to complete with a COMMIT only, and so assumes that transactions = 'user commits.' For this reason, comparing per transaction statistics between Statspack and BSTAT/ESTAT can result in significantly different per transaction ratios.

> **Note:** If you choose to run BSTAT/ESTAT in conjunction to Statspack, do not run both as the same user, because there is a table name conflict: table STATS$WAITSTAT.

How Statspack Works

Statspack is a set of SQL, PL/SQL, and SQL*Plus scripts that allow the collection, automation, storage, and viewing of performance data. A user is automatically created by the installation script. This user, PERFSTAT, owns all objects needed by this package. This user is granted limited query-only privileges on the V$ views required for performance tuning.

Statspack users become familiar with the concept of a snapshot, a single collection of performance data. Each snapshot taken is identified by a snapshot ID, which is a unique number generated at the time the snapshot is taken. Each time a new collection is taken, a new SNAP_ID is generated.

The SNAP_ID, along with the database identifier (DBID) and instance number (INSTANCE_NUMBER), comprise the unique key for a snapshot (using this unique combination allows storage of multiple instances of an Oracle Real Application Clusters database in the same tables).

After snapshots are taken, it is possible to run the performance report. The performance report prompts for the two snapshot IDs the report will process. The report produced calculates the activity on the instance between the two snapshot periods specified, similar to the BSTAT/ESTAT report. To compare, the first SNAP_ID supplied can be considered the equivalent of running BSTAT; the second SNAP_ID specified can be considered the equivalent of ESTAT. Unlike BSTAT/ESTAT, which can by it's nature only compare two static data points, the report can compare any two snapshots specified.

Configuring Statspack

Database Space Requirements for Statspack

The amount of database space required by the package varies considerably based on the frequency of snapshots, the size of the database and instance, and the amount of data collected (which is configurable). It is, therefore, difficult to provide general storage clauses and space utilization predictions that are accurate at each site.

Note: The default initial and next extent sizes are 100k, 1MB, or 5MB for all Statspack tables and indexes. Approximately 64MB is required to install Statspack.

Statspack in Dictionary Managed Tablespaces

If you install the package in a dictionary-managed tablespace, then monitor the space used by the objects created and, if required, adjust the storage clauses of the segments.

Statspack in Locally Managed Tablespaces

If you install the package in a locally-managed tablespace, then storage clauses are not required, because the storage characteristics are automatically managed.

Installing Statspack

There are two ways to install Statspack:

- Interactive Statspack Installation
- Batch Mode Statspack Installation

Batch mode is useful when you do not want to be prompted for the PERFSTAT user's default and temporary tablespaces.

Interactive Statspack Installation

The first step is to create the PERFSTAT user, which owns all PL/SQL code and database objects created, including the Statspack tables, constraints, and the Statspack package. During installation, you are prompted for the PERFSTAT user's default and temporary tablespaces. The default tablespace is used to create all Statspack objects, such as tables and indexes. The temporary tablespace is used for sort-type activities.

> **See Also:** *Oracle9i Database Concepts* for more information on temporary tablespaces

Note:

- Do not specify the SYSTEM tablespace for the PERFSTAT user's DEFAULT or TEMPORARY tablespaces. If SYSTEM is specified, the installation aborts with an error specifying the problem. Oracle Corporation does not recommend using the SYSTEM tablespace to store statistics data or for sorting. Use a TOOLS tablespace to store the data, and use your instance's TEMPORARY tablespace for sorting. To recover from this error, run the de-install (spdrop.sql) script, then rerun the installation.
- During installation, the DBMS_SHARED_POOL and DBMS_JOB PL/SQL packages are created. DBMS_SHARED_POOL pins the Statspack package in the shared pool. DBMS_JOB is created on the assumption that you want to schedule periodic snapshots automatically using DBMS_JOB.

To install the package, either change to the ORACLE_HOME rdbms/admin directory, or fully specify the ORACLE_HOME/rdbms/admin directory when calling the installation script, SPCREATE.

To run the installation script, you must use SQL*Plus and connect as a user with SYSDBA privilege. For example, start SQL*Plus, then:

On Unix:

```
SQL>  CONNECT / AS SYSDBA
SQL>  @?/rdbms/admin/spcreate
```

On NT:

```
SQL>  CONNECT / AS SYSDBA
SQL>  @%ORACLE_HOME%\rdbms\admin\spcreate
```

The SPCREATE install script runs three other scripts. These scripts are called automatically, so you do not need to run them:

- SPCUSR: Creates the user and grants privileges
- SPCTAB: Creates the tables
- SPCPKG: Creates the package

Check each of the three output files produced (SPCUSR.LIS, SPCTAB.LIS, SPCPKG.LIS) by the installation to ensure that no errors were encountered.

Batch Mode Statspack Installation

To install in batch mode, you must assign values to the SQL*Plus variables that specify the default and temporary tablespaces before running SPCREATE. The variables are:

- DEFAULT_TABLESPACE: For the default tablespace
- TEMPORARY_TABLESPACE: For the temporary tablespace

For example, on Unix:

```
SQL>  CONNECT / AS SYSDBA
SQL>  define default_tablespace='tools'
SQL>  define temporary_tablespace='temp'
SQL>  @?/rdbms/admin/spcreate
```

SPCREATE no longer prompts for the above information.

Note: After the setup is complete, you can change the password of the PERFSTAT user for security purposes.

Using Statspack

Taking a Statspack Snapshot

The simplest interactive way to take a snapshot is to login to SQL*Plus as the PERFSTAT user and run the procedure STATSPACK.SNAP. For example:

```
SQL>  CONNECT perfstat/perfstat
SQL>  EXECUTE statspack.snap;
```

Note: In an Oracle Real Application Clusters environment, you must connect to the instance you want to collect data for.

This stores the current values for the performance statistics in the Statspack tables, and can be used as a baseline snapshot for comparison with another snapshot taken at a later time.

For better performance analysis, set the initialization parameter TIMED_STATISTICS to true. This way, Statspack data collected includes important timing information. The TIMED_STATISTICS parameter can be dynamically changed using the ALTER SYSTEM statement. Timing data is important and ususally is required by Oracle support to diagnose performance problems.

Typically, to automate the gathering and reporting phases (such as during a benchmark), you might need to know the snap_id of the snapshot just taken. To take a snapshot and display the snap_id, call the STATSPACK.SNAP function. Below is an example of calling the snap function using an anonymous PL/SQL block in SQL*Plus:

```
SQL> variable snap number;
SQL> begin   :snap := statspack.snap;   end;
  2  /
PL/SQL procedure successfully completed.
SQL> print snap
      SNAP
----------
        12
```

Automating Statistics Gathering

To make performance comparisons from one day, week, or year to the next, there must be multiple snapshots taken over a period of time. The best method to gather snapshots is to automate the collection on a regular time interval. There are the following options:

- Within the database, use the Oracle DBMS_JOB procedure to schedule snapshots.
- Use operating system utilities such as 'cron' on Unix or 'at' on NT to schedule snapshots.

Using DBMS_JOB to Collect Statistics

To use an Oracle-automated method for collecting statistics, use the DBMS_JOB package. A sample script on how to do this is supplied in SPAUTO.SQL, which schedules a snapshot every hour, on the hour.

You might want to schedule snapshots at regular times each day to reflect your system's OLTP or batch peak loads. For example, take snapshots at 9am, 10am,

11am, 12 midday, and 6pm for the OLTP load, then a snapshot at 12 midnight and another at 6am for the batch window.

In order to use DBMS_JOB to schedule snapshots, the JOB_QUEUE_PROCESSES initialization parameter must be set to greater than 0 in the init.ora file for the job to be run automatically.

Example of an initialization entry:

```
# Set to enable the job queue process to start. This allows DBMS_JOB to
# schedule automatic statistics collection using Statspack
JOB_QUEUE PROCESSES=1
```

If using SPAUTO.SQL in an Oracle Real Application Clusters environment, the SPAUTO.SQL script must be run once on each instance in the cluster. Similarly, the JOB_QUEUE_PROCESSES parameter must also be set for each instance.

Changing the Interval of Statistics Collection

Use the DBMS_JOB.INTERVAL procedure to change the interval of statistics collection. For example:

```
EXECUTE DBMS_JOB.INTERVAL(1,'SYSDATE+(1/48)');
```

Where 'SYSDATE+(1/48)' results in the statistics being gathered each 1/48 hours (that is, every half hour).

To force the job to run immediately:

```
EXECUTE DBMS_JOB.RUN(<job number>);
```

To remove the autocollect job:

```
EXECUTE DBMS_JOB.REMOVE(<job number>);
```

> **See Also:** *Oracle9i Supplied PL/SQL Packages Reference* for more information on the DBMS_JOB package

Running a Statspack Performance Report

After snapshots are taken, it is possible to generate a performance report. The SQL script that generates the report prompts for the two snapshot ID's to be processed.

There are two reports. The first (spreport.sql) is a general instance health report, covering all aspects of instance performance. The second report, a SQL report, usually is run after examining the first report. The SQL report only reports on a

single SQL statement (as identified by the hash value). Both reports prompt for the beginning snapshot ID, the ending snapshot ID, and the report name. The instance report then calculates and prints ratios, increases, and so on for all statistics between the two snapshot periods, similar to the BSTAT/ESTAT report. The SQL report only reports on data relating to the single SQL statement.

> **Note:** It is not correct to specify begin and end snapshots where the begin snapshot and end snapshot were taken from different instance startups. In other words, the instance must not have been shutdown between the times that the begin and end snapshots were taken.
>
> This is necessary because the database's dynamic performance tables, which Statspack queries to gather the data, are memory resident. Hence, shutting down the database resets the values in the performance tables to 0. Because Statspack subtracts the begin-snapshot statistics from the end-snapshot statistics, the resulting output is invalid. If begin and end snapshots taken between shutdowns are specified in the report, then the report shows an appropriate error to indicate this.

Separating the phase of data gathering from producing a report allows the flexibility of basing a report on any data points selected. For example, it might be reasonable for the DBA to use the supplied automation script to automate data collection every hour, on the hour. If, at some later point, a performance issue arose that might be better investigated by looking at a three hour data window rather than an hour's worth of data, then all the DBA needs to do is specify the required start point and end point when running the report.

Running the Statspack Report

To examine the change in instance-wide statistics between two time periods, the SPREPORT.SQL file is run while connected to the PERFSTAT user. The SPREPORT.SQL command file is located in the `rdbms/admin` directory of the Oracle home.

> **Note:** In an Oracle Real Application Clusters environment, you must connect to the instance you want to report on.

You are prompted for the following:

1. The beginning snapshot ID
2. The ending snapshot ID
3. The name of the report text file to be created

> **Note:** Blank lines between lines of snapshot IDs means that the instance has been restarted (shutdown/startup) between those times. This helps identify which begin and end snapshots can be used together when running a Statspack report.

For example, on Unix:

```
SQL>  connect perfstat/perfstat
SQL>  @?/rdbms/admin/spreport
```

For example, on NT:

```
SQL>  connect perfstat/perfstat
SQL>  @%ORACLE_HOME%\rdbms\admin\spreport
```

Example output:

```
SQL>  connect perfstat/perfstat
Connected.
SQL>  @spreport

DB Id    DB Name      Inst Num Instance
----------- ------------ -------- ------------
2618106428 PRD1                1 prd1
Completed Snapshots
                           Snap                        Snap
Instance     DB Name         Id   Snap Started    Level Comment
------------ ------------ ----- ----------------- ----- ----------------
prd1         PRD1             1 11 May 2000 12:07     5
                              2 11 May 2000 12:08     5

Specify the Begin and End Snapshot Ids
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Enter value for begin_snap: 1
Begin Snapshot Id specified: 1

Enter value for end_snap: 2
End   Snapshot Id specified: 2
```

```
Specify the Report Name
~~~~~~~~~~~~~~~~~~~~~~~
The default report file name is sp_1_2  To use this name, press <return> to
continue, otherwise enter an alternative. Enter value for report_name:
<press return or enter a new name>
Using the report name sp_1_2
```

The report now scrolls past and is also written to the file specified (for example, `sp_1_2.lis`).

To run a report without being prompted, assign values to the SQL*Plus variables that specify the begin snap ID, the end snap ID, and the report name before running `SPREPORT`.

The variables are:

- `BEGIN_SNAP`: specifies the begin snapshot ID
- `END_SNAP`: specifies the end snapshot ID
- `REPORT_NAME`: specifies the report output name

For example, on Unix:

```
SQL>  connect perfstat/perfstat
SQL>  define begin_snap=1
SQL>  define end_snap=2
SQL>  define report_name=batch_run
SQL>  @?/rdbms/admin/spreport
```

`SPREPORT` no longer prompts for the above information.

Running the SQL Report

After the instance report has been examined, often there are high-load SQL statements when should be examined more closely. The SQL report `sprepsql.sql`, displays statistics, the complete SQL text, and (if level six snapshot has been taken), information on any SQL plan(s) associated with that statement.

The SQL statement to be reported on is identified by a hash value, which is a numerical representation of the statement's SQL text. The hash value for each statement is displayed for each statement in the SQL sections of the instance report.

The `sprepsql.sql` file is executed while connected to the `PERFSTAT` user. It is located in the `rdbms/admin` directory of the Oracle home.

Note: In an Oracle Real Application Clusters environment, you must connect to the instance you want to report on.

You are prompted for the following:

1. The beginning snapshot ID
2. The ending snapshot ID
3. The hash value for the SQL statement
4. The name of the report text file to be created

Example output:

```
SQL>  connect perfstat/perfstat
Connected.
SQL>  @spreport


   DB Id    DB Name      Inst Num Instance
----------- ------------ -------- ------------
 2618106428 PRD1                1 prd1


Completed Snapshots

                           Snap                     Snap
Instance     DB Name         Id   Snap Started    Level Comment
------------ ------------ ----- ----------------- ----- ----------------------
prd1         PRD1            37 02 Mar 2001 11:01     6
                             38 02 Mar 2001 12:01     6

                             39 08 Mar 2001 09:01     5
                             40 08 Mar 2001 10:02     5

Specify the Begin and End Snapshot Ids
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Enter value for begin_snap: 39
Begin Snapshot Id specified: 39

Enter value for end_snap: 40
End   Snapshot Id specified: 40
```

```
Specify the Hash Value
~~~~~~~~~~~~~~~~~~~~~~
Enter value for hash_value: 1988538571
Hash Value specified is: 1988538571


Specify the Report Name
~~~~~~~~~~~~~~~~~~~~~~~
The default report file name is sp_39_40_1988538571.  To use this name,
press <return> to continue, otherwise enter an alternative.
Enter value for report_name:

Using the report name sp_39_40_1988538571

  The report will scroll past, and also be written to the file
  specified (e.g. sp_39_40_1988538571.lis).
```

Similar to `spreport.sql`, the SQL report can be run in batch mode. To run a report without being prompted, assign values to the SQL*Plus variables that specify the begin snap ID, the end snap ID, and the report name before running `spreport`.

The variables are:

- `BEGIN_SNAP`: specifies the begin snapshot ID
- `END_SNAP`: specifies the end snapshot ID
- `HASH_VALUE`: specifies the hash value
- `REPORT_NAME`: specifies the report output name

For example:

```
SQL>  connect perfstat/perfstat
SQL>  define begin_snap=39
SQL>  define end_snap=40
SQL   define hash_value=1988538571
SQL>  define report_name=batch_sql_run
SQL>  @sprepsql
```

`SPREPSQL` no longer prompts for the above information.

Gathering Optimizer Statistics on the PERFSTAT Schema

For best performance when running the performance reports, collect optimizer statistics for tables and indexes owned by PERFSTAT. This should be performed whenever there is significant change in data volumes in PERFSTAT's tables. To do

this, use DBMS_STATS or DBMS_UTILITY, and specify the PERFSTAT user: For example:

```
EXECUTE DBMS_STATS.GATHER_SCHEMA_STATS(OWNNAME=>'PERFSTAT',CASCADE=>TRUE);
```

or

```
EXECUTE DBMS_UTILITY.ANALYZE_SCHEMA('PERFSTAT','COMPUTE');
```

Configuring the Amount of Data Captured in Statspack

Both the snapshot level and the thresholds specified affect the amount of data Statspack captures.

It is possible to change the amount of information gathered by the package by specifying a different snapshot 'level'. In other words, the level chosen (or defaulted) decides the amount of data collected. The higher the snapshot level, the more data is gathered. The default level set by the installation is level 5.

For typical usage, level 5 snapshot is effective on most sites. There are certain situations when using a level 6 snapshot is beneficial. These include the following:

- When taking your first baseline
- When a new application or an application change is installed (that is, taking a new baseline)
- After gathering optimizer statistics

See Also: "Snapshot Levels" on page 21-16

Snapshot SQL Thresholds

There are other parameters that can be configured in addition to the snapshot level. These parameters are used as thresholds when collecting data on SQL statements; data is captured on any SQL statements that breach the specified thresholds. Snapshot level and threshold information used by the package is stored in the STATS$STATSPACK_PARAMETER table.

Changing the Default Values for Snapshot Levels and SQL Thresholds

You can change the default parameters used for taking snapshots so that they are tailored to the instance's workload. The full list of parameters that can be passed into the MODIFY_STATSPACK_PARAMETER procedure are the same as those for the SNAP procedure.

See Also: "Parameters for SNAP and MODIFY_STATSPACK_PARAMETERS Procedures" on page 21-17

Temporarily Using New Values To temporarily use a snapshot level or threshold that is different than the instance's default snapshot values, simply specify the required threshold or snapshot level when taking the snapshot. This value is used only for the immediate snapshot taken; the new value is not saved as the default.

For example, take a single level 6 snapshot (do not save level 6 as the default):

```
SQL>  EXECUTE STATSPACK.SNAP(i_snap_level=>6);
```

Saving New Defaults You can save the new value as the instance's default in two ways:

1. Take a snapshot, and specify the new defaults to be saved to the database (using STATSPACK.SNAP the I_MODIFY_PARAMETER input variable).

   ```
   SQL>  EXECUTE STATSPACK.SNAP -
         (i_snap_level=>10, i_modify_parameter=>'true');
   ```

   Setting the I_MODIFY_PARAMETER value to `true` saves the new thresholds in the STATS$STATSPACK_PARAMETER table. These thresholds are used for all subsequent snapshots.

   If the I_MODIFY_PARAMETER was set to `false` or if it were omitted, then the new parameter values would not be saved. Only the snapshot taken at that point uses the specified values. Any subsequent snapshots use the preexisting values in the STATS$STATSPACK_PARAMETER table.

2. Change the defaults immediately without taking a snapshot using the STATSPACK.MODIFY_STATSPACK_PARAMETER procedure. For example, to change the snapshot level to 10, and the SQL thresholds for BUFFER_GETS and DISK_READS, the following statement can be issued:

   ```
   SQL>  EXECUTE STATSPACK.MODIFY_STATSPACK_PARAMETER -
         i_snap_level=>10, i_buffer_gets_th=>10000, i_disk_reads_th=>1000);
   ```

   This procedure changes the values permanently, but does not take a snapshot.

See Also: "Parameters for SNAP and MODIFY_STATSPACK_PARAMETERS Procedures" on page 21-17

Snapshot Levels

Levels >= 0 General Performance Statistics This level and any level greater than 0 collects general performance statistics, such as wait statistics, system events, system statistics, rollback segment data, row cache, SGA, background events, session events, lock statistics, buffer pool statistics, and parent latch statistics.

Levels >= 5 Additional Data: SQL Statements This level includes all statistics gathered in the lower level(s), and additionally gathers the performance data on high resource usage SQL statements. In a level 5 snapshot (or above), the time required for the snapshot to complete is dependant on the `SHARED_POOL_SIZE` and on the number of SQL statements in the shared pool at the time the snapshot is taken. The larger the shared pool, the longer the time taken to complete the snapshot.

The SQL statements gathered by Statspack are those that exceed one of six predefined threshold parameters:

- Number of executions of the SQL statement (default 100)
- Number of disk reads performed by the SQL statement (default 1,000)
- Number of parse calls performed by the SQL statement (default 1,000)
- Number of buffer gets performed by the SQL statement (default 10,000)
- Size of sharable memory used by the SQL statement (default 1m)
- Version count for the SQL statement (default 20)

The values of each of these threshold parameters are used when deciding which SQL statements to collect. If a SQL statement's resource usage exceeds any one of the above threshold values, then it is captured during the snapshot.

The SQL threshold levels used are either those stored in the table `STATS$STATSPACK_PARAMETER` or by the thresholds specified when the snapshot is taken.

Levels >= 6 Additional Data: SQL Plans and SQL Plan Usage This level includes all statistics gathered in the lower level(s). Additionally, it gathers SQL plans and plan usage data for each of the high resource usage SQL statements captured.

A level 6 snapshot gathers valuable information for determining whether the execution plan used for a SQL statement has changed. Therefore, level 6 snapshots should be used whenever there is the possibility that a plan may change.

To gather the plan for a SQL statement, the statement must be in the shared pool at the time the snapshot is taken, and it must exceed one of the SQL thresholds. To

gather plans for all statements in the shared pool, specify the executions threshold to be zero (0) for those snapshots.

> **See Also:** "Changing the Default Values for Snapshot Levels and SQL Thresholds" on page 21-14 for information on how to do this

Levels >= 10 Additional Statistics: Parent and Child Latches This level includes all statistics gathered in the lower levels. Additionally, it gathers parent and child latch information. Sometimes, data gathered at this level can cause the snapshot to take longer to complete. This level can be resource-intensive, and it should only be used when advised by Oracle personnel.

Specifying a Session ID

If you want to gather session statistics and wait events for a particular session (in addition to the instance statistics and wait events), specify the session ID in the call to Statspack. The statistics gathered for the session include session statistics, session events, and lock activity. The default behavior is to not gather session level statistics.

For example:

```
SQL>  EXECUTE STATSPACK.SNAP(i_session_id=>3);
```

Parameters for SNAP and MODIFY_STATSPACK_PARAMETERS Procedures

Parameters able to be passed in to the STATSPACK.SNAP and STATSPACK.MODIFY_STATSPACK_PARAMETER procedures are as follows:

```
                   Range of      Default
Parameter Name     Valid Values  Value    Meaning
-----------------  ------------  -------  ---------------------------------
i_snap_level       0, 5, 6, 10   5        Snapshot Level
i_ucomment         Text          Blank    Comment to be stored with Snapshot
i_executions_th    Integer >=0   100      SQL Threshold: number of times
                                          the statement was executed
i_disk_reads_th    Integer >=0   1,000    SQL Threshold: number of disk reads
                                          the statement made
i_parse_calls_th   Integer >=0   1,000    SQL Threshold: number of parse
                                          calls the statement made
i_buffer_gets_th   Integer >=0   10,000   SQL Threshold: number of buffer
                                          gets the statement made
i_sharable_mem_th  Integer >=0   1048576  SQL Threshold: amount of sharable
                                          memory
i_version_count_th Integer >=0   20       SQL Threshold: number of versions
                                          of a SQL statement
i_session_id       Valid sid     0 (no    Session Id of the Oracle Session
                   from          session) to capture session granular
                   v$session               statistics for
i_modify_parameter True,False    False    Save the parameters specified for
                                          future snapshots?
```

Time Units Used for Wait Events

Oracle supports capturing certain performance data with microsecond granularity. Views that include microsecond timing include the following:

- V$SESSION_WAIT, V$SYSTEM_EVENT, V$SESSION_EVENT (TIME_WAITED_MICRO column)
- V$SQL (CPU_TIME, ELAPSED_TIME columns)
- V$LATCH (WAIT_TIME column)
- V$SQL_WORKAREA, V$SQL_WORKAREA_ACTIVE (ACTIVE_TIME column)

> **Note:** Existing columns in other views continue to use centisecond times.

Because microsecond timing might not be appropriate for rolled-up data, such as that displayed by Statspack, Statspack displays most times in seconds, and average times in milliseconds (for easier comparison with operating system monitoring utilities which often report timings in milliseconds).

For clarity, the time units used are specified in the column headings of each timed column in the Statspack report. The following convention used is:

(s) - a second

(cs) - a centisecond (100th of a second)

(ms) - a millisecond (1,000th of a second)

(us) - a microsecond (1,000,000th of a second)

Event Timings

If timings are available, the Statspack report orders wait events by time (in the Top-5 and background and foreground wait events sections).

If `TIMED_STATISTICS` is `false` for the instance, but a subset of users or programs set `TIMED_STATISTICS` (set to `true` dynamically), then the Statspack report output can look inconsistent, where some events have timings (those which the individual programs / users waited for) and some do not. The Top-5 section also looks unusual in this situation.

Optimally, `TIMED_STATISTICS` should be set to `true` at the instance level for ease of diagnosing performance problems.

Managing and Sharing Statspack Performance Data

Sharing Data Through Export

If you want to share data with other sites (for example, if Oracle Support requires the raw statistics), then it is possible to export the `PERFSTAT` user. An export parameter file (`SPUEXP.PAR`) is supplied for this purpose. To use this file, supply the export command with the userid parameter, along with the export parameter file name. For example:

```
exp userid=perfstat/perfstat parfile=spuexp.par
```

This creates a file called `SPUEXP.DMP` and the log file `SPUEXP.LOG`. If you want to load the data into another database, use the import command.

> **See Also:** *Oracle9i Database Utilities* for more information on using export and import

Removing Unnecessary Data

Purge unnecessary data from the PERFSTAT schema using the SPPURGE.SQL script. This deletes snapshots that fall between the begin and end range of the snapshot ID's specified.

Note: You should export the schema as a backup before running this script, either using your own export parameters or those provided in SPUEXP.PAR.

Purging can require the use of a large rollback segment, because all data relating to each snapshot ID to be purged is deleted. To avoid rollback segment extension errors, explicitly use a large rollback segment. This can be done by executing the SET TRANSACTION USE ROLLBACK SEGMENT statement before running the SPPURGE.SQL script. Alternatively, to avoid rollback segment extension errors, specify a smaller range of snapshot ID's to purge.

See Also: *Oracle9i SQL Reference*

When SPPURGE is run, the instance currently connected to and the available snapshots are displayed. The DBA is then prompted for the low snap ID and high snap ID. All snapshots that fall within this range are purged.

For example, connect to PERFSTAT using SQL*Plus, then:

```
SQL>  CONNECT perfstat/perfstat
SQL>  SET TRANSACTION USE ROLLBACK SEGMENT rbig;
SQL>  @sppurge

   Database Instance currently connected to
   ========================================
                                  Instance
      DB Id    DB Name    Inst Num Name
   ----------- ---------- -------- ----------
     720559826 PERF              1 perf

   Snapshots for this database instance
   ====================================
               Snap
```

```
   Snap Id Level Snapshot Started      Host            Comment
---------- ----- -------------------- --------------- -------------------
         1     5  30 Feb 2000 10:00:01 perfhost
         2     5  30 Feb 2000 12:00:06 perfhost
         3     5  01 Mar 2000 02:00:01 perfhost
         4     5  01 Mar 2000 06:00:01 perfhost
```

Caution: SPPURGE.SQL deletes all snapshots ranging between the lower and upper bound snapshot IDs specified for the database instance connected to. You might want to export this data before continuing.

```
Specify the Low Snap ID and High Snap ID range to purge
   ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

   Enter value for losnapid: 1
   Using 1 for lower bound.

   Enter value for hisnapid: 2
   Using 2 for upper bound.

   Deleting snapshots 1 - 2
```

Purge of specified snapshot range complete. If you want to rollback the purge, it is still possible to do so. Exiting from SQL*Plus automatically commits the purge.

```
SQL> -- end of example output
```

To purge in batch mode, you must assign values to the SQL*Plus variables that specify the low and high snapshot IDs to purge. The variables are:

- LOSNAPID: Begin snapshot ID
- HISNAPID: End snapshot ID

For example:

```
SQL>  CONNECT perfstat/perfstat
SQL>  DEFINE losnapid=1
SQL>  DEFINE hisnapid=2
SQL>  @sppurge
```

SPPURGE no longer prompts for the above information.

Truncating All Statspack Data

If you want to truncate all performance data indiscriminately, use SPTRUNC.SQL. This script truncates all statistics data gathered.

> **Note:** Oracle Corporation recommends that you export the schema as a backup before running this script, either using your own export parameters or those provided in SPUEXP.PAR.

For example, connect to PERFSTAT using SQL*Plus, then:

```
SQL>  CONNECT perfstat/perfstat
SQL>  @sptrunc
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

> **Note:** Running SPTRUNC.SQL removes *all* data from Statspack tables. You might want to export the data before continuing.

```
If you would like to continue, enter any string, followed by <return>.
Enter value for anystring:
entered - starting truncate operation
Table truncated.
<etc>
Truncate operation complete.
```

Oracle Real Application Clusters Considerations with Statspack

The unique identifiers for a database instance used by Statspack are the DBID and the INSTANCE_NUMBER. When using Oracle Real Application Clusters, the INSTANCE_NUMBER could change between startups (either because the INSTANCE_NUMBER parameter is set in the initialization file or because the instances are started in a different order).

In this case, because Statspack uses the INSTANCE_NUMBER and the DBID to identify the instance's snapshot preferences, this could inadvertently result in a different set of levels or thresholds being used when snapshotting an instance.

There are three conditions that must be met for this to occur:

- The instance numbers must have switched between startups.
- The DBA must have modified the default Statspack parameters used for at least one of the instances.
- The parameters used (for example, thresholds and snapshot level) must not be the same on all instances.

The only way the parameters differ is if the parameters have been explicitly modified by the DBA after installation, either by saving the specified values or by using the MODIFY_STATSPACK_PARAMETER procedure. Check whether any of the Statspack snapshot parameters are different for the instances by querying the STATS$STATSPACK_PARAMETER table.

Note: If you have changed the default Statspack parameters, you can avoid encountering this problem by hard-coding the INSTANCE_NUMBER in the initialization parameter file for each of the instances in the Oracle Real Application Clusters database. For recommendations and issues with setting the INSTANCE_NUMBER initialization parameter, see the Oracle Real Application Clusters documentation.

Removing Statspack

To deinstall Statspack, connect as a user with SYSDBA privilege and run the following SPDROP script from SQL*Plus. For example:

```
SQL>  CONNECT / AS SYSDBA
SQL>  @spdrop
```

This script calls two other scripts:

- SPDTAB -> Drops tables and public synonyms
- SPDUSR -> Drops the user

Check each of two output files produced (SPDTAB.LIS, SPDUSR.LIS) to ensure that the package was completely deinstalled.

Statspack Supplied Scripts

Scripts for Statspack Installation

The following must be run as a user with SYSDBA privilege:

- SPCREATE.SQL: Creates entire Statspack environment (calls SPCUSR.SQL, SPCTAB.SQL, SPCPKG.SQL)
- SPDROP.SQL: Drops entire Statspack environment (calls SPDTAB.SQL, SPDUSR.SQL)

The following are run as a user with SYSDBA privilege by the calling scripts (above):

- SPDTAB.SQL: Drops Statspack tables
- SPDUSR.SQL: Drops the Statspack user (PERFSTAT)

The following are run as PERFSTAT by the calling scripts (above):

- SPCUSR.SQL: Creates the Statspack user (PERFSTAT)
- SPCTAB.SQL: Creates Statspack tables
- SPCPKG.SQL: Creates the Statspack package

Scripts for Statspack Reporting and Automation

The following must be run as PERFSTAT:

- SPREPORT.SQL: Generates a Statspack report
- SPREPSQL.SQL: Generates a Statspack SQL report for the specific SQL hash value specified
- SPREPINS.SQL: Generates a Statspack report for the database and instance specified
- SPAUTO.SQL: Automates Statspack statistics collection (using DBMS_JOB)

Scripts for Upgrading Statspack

The following must be run as PERFSTAT:

SPUP817.SQL: Converts data from the 8.1.7 schema to the 9.0 schema. Backup the existing schema before running the upgrade. If upgrading from Statspack 8.1.6, SPUP816.SQL must be run first.

Scripts for Statspack Performance Data Maintenance

The following must be run as PERFSTAT:

- SPPURGE.SQL: Purges a limited range of Snapshot ID's for a given database instance.
- SPTRUNC.SQL: Truncates all performance data in Statspack tables

> **Caution:** Do not use unless you wan to remove all data in the schema you are using. You can choose to export the data as a backup before using this script.

- SPUEXP.PAR: An export parameter file supplied for exporting the whole PERFSTAT user.

Scripts for Statspack Documentation

The SPDOC.TXT file contains instructions and documentation on the Statspack package.

Statspack Limitations

Statspack is not supported with releases earlier than 8.1.6.

Storing data from multiple databases in one PERFSTAT user account is currently not supported.

Part V

Optimizing Instance Performance

Part V describes how to tune various elements of your database system to optimize performance of an Oracle instance.

The chapters in this part are:

- Chapter 22, "Instance Tuning"
- Chapter 23, "Tuning Networks"

22

Instance Tuning

This chapter discusses the method used for performing tuning. This chapter contains the following sections:

- Performance Tuning Principles
- Performance Tuning Steps
- Interpreting Oracle Statistics
- Wait Events
- Idle Wait Events

Performance Tuning Principles

Performance tuning requires a different, although related, method to the initial configuration of a system. Configuring a system involves allocating resources in an ordered manner so that the initial system configuration is functional.

Tuning is driven by identifying the *most significant bottleneck* and making the appropriate changes to reduce or eliminate the effect of that bottleneck. Usually, tuning is performed reactively, either while the system is preproduction or after it is live.

Baselines

The most effective way to tune is to have an established performance baseline that can be used for comparison if a performance issue arises. Most DBAs know their system well and can easily identify peak usage periods. For example, the peak periods could be between 10.00am and 12.00pm and also between 1.30pm and 3.00pm. This could include a batch window of 12.00am midnight to 6am.

It is important to identify these high-load times at the site and install a monitoring tool that gathers performance data for those times. Optimally, data gathering should be configured from when the application is in its initial trial phase (during the QA cycle). Otherwise, this should be configured when the system is first in production.

Note: Oracle recommends using the Enterprise Manager (EM) Diagnostics Pack for systems monitoring and tuning due to its extended feature list. However, if your site does not have EM, then Statspack can be used to gather Oracle instance statistics.

For illustration purposes, a combination of Statspack report output and direct queries from the V$ views are used in examples, because they are available on all installations.

See Also: Chapter 20, "Oracle Tools to Gather Database Statistics" for detailed information on Oracle instance performance tuning tools

Ideally, baseline data gathered should include the following:

- Application statistics (transaction volumes, response time)
- Database statistics
- OS statistics
- Disk I/O statistics
- Network statistics

The Symptoms and the Problems

A common pitfall in performance tuning is to mistake the symptoms of a problem for the actual problem itself. It is important to recognize that many performance statistics indicate the symptoms, and that identifying the symptom is not sufficient data to implement a remedy. For example:

- Slow physical I/O

  Generally, this is caused by poorly-configured disks. However, it could also be caused by a significant amount of unnecessary physical I/O on those disks issued by poorly-tuned SQL.

- Latch contention

  Rarely is latch contention tunable by reconfiguring the instance. Rather, latch contention usually is resolved through application changes.

- Excessive CPU usage

  Excessive CPU usage usually means that there is little idle CPU on the system. This could be caused by an inadequately-sized system, by untuned SQL statements, or by inefficient application programs.

See Also: Table 22–1, "Wait Events and Potential Causes" on page 22-15

When to Tune

There are two distinct types of tuning: proactive monitoring and bottleneck elimination.

Proactive Monitoring

Proactive monitoring usually occurs on a regularly scheduled interval, where a number of performance statistics are examined to identify whether the system

behavior and resource usage has changed. Proactive monitoring also can be called proactive tuning.

Usually, monitoring does not result in configuration changes to the system, unless the monitoring exposes a serious problem that is developing. In some situations, experienced performance engineers can identify potential problems through statistics alone, although accompanying performance degradation is usual.

'Tweaking' a system when there is no apparent performance degradation as a proactive action can be a dangerous activity, resulting in unnecessary performance drops. Tweaking a system should be considered reactive tuning, and the steps for reactive tuning should be followed.

Monitoring is usually part of a larger capacity planning exercise, where resource consumption is examined to see the changes in the way the application is being used and the way the application is using the database and host resources.

Bottleneck Elimination: Tuning

Tuning usually implies fixing a performance problem. However, tuning should be part of the lifecycle of an application, through the analysis, design, coding, production, and maintenance stages. Many times, the tuning phase is left until the system is in production. At this time, tuning becomes a reactive fire-fighting exercise, where the most important bottleneck is identified and fixed.

Usually, the purpose for tuning is to reduce resource consumption or to reduce the elapsed time for an operation to complete. Either way, the goal is to improve the effective use of a particular resource. In general, performance problems are caused by the over-use of a particular resource. That resource is the *bottleneck* in the system. There are a number of distinct phases in identifying the bottleneck and the potential fixes. These are discussed below.

Remember that the different forms of contention are symptoms that can be fixed by making changes in the following places:

- Changes in the application, or the way the application is used
- Changes in Oracle
- Changes in the host hardware configuration

Often, the most effective way of resolving a bottleneck is to change the application.

Performance Tuning Steps

Below are the main steps in the Oracle Performance Method:

1. Get candid feedback from users about the scope of the performance problem. This step is to Define the Problem.
2. Obtain a full set of OS, database, and application statistics. Then Examine the Host System and Examine the Oracle Statistics for any evidence.
3. Consider the list of common performance errors to see whether the data gathered suggests that they are contributing to the problem.
4. Build a conceptual model of what is happening on the system using the performance data gathered.
5. Propose changes to be made and the expected result of implementing the changes. Then, Implement and Measure Change in application performance.
6. Determine whether the performance objective defined in step 1 has been met. If not, then repeat steps 5 and 6 until the performance goals are met.

> **See Also:** *Oracle9i Database Performance Methods* for a list of common errors and for a theoretical description of this performance method

The remainder of this chapter covers the steps of the Oracle performance method in detail.

Define the Problem

It is vital to develop a good understanding of the purpose of the tuning exercise and the nature of the problem before attempting to implement a solution. Without this understanding, it is virtually impossible to implement effective changes. The data gathered during this stage helps determine the next step to take and what evidence to examine.

Gather the following data:

1. Identify the performance objective.

   What is the measure of acceptable performance? How many transactions per hour, or seconds, response time will meet the required performance level?

2. Identify the scope of the problem.

   What is affected by the slowdown? For example, is the whole instance slow? Is it a particular application, program, specific operation, or a single user?

3. Identify the time frame when the problem occurs.

   Is the problem only evident during peak hours? Does performance deteriorate over the course of the day? Was the slowdown gradual (over the space of months or weeks) or sudden?

4. Quantify the slowdown.

   This helps identify the extent of the problem and also acts as a measure for comparison when deciding whether changes implemented to fix the problem have actually made an improvement. Find a consistently reproducible measure of the response time or job run time. How much worse are the timings than when the program was running well?

5. Identify any changes.

   Identify what has changed since performance was acceptable. This may narrow the potential cause quickly. For example, has the operating system software, hardware, application software, or Oracle release been upgraded? Has more data been loaded into the system, or has the data volume or user population grown?

At the end of this phase, you should have a good understanding of the symptoms. If the symptoms can be identified as local to a program or set of programs, then the problem is handled in a different manner than instance-wide performance issues.

> **See Also:** Chapter 6, "Optimizing SQL Statements" for information on solving performance problems specific to an application or user

Examine the Host System

Look at the load on the database server, as well as the database instance. Consider the operating system, the I/O subsystem, and network statistics, because examining these areas helps determine what might be worth further investigation. In multitier systems, also examine the application server middle-tier hosts.

Examining the host hardware often gives a strong indication of the bottleneck in the system. This determines which Oracle performance data could be useful for cross-reference and further diagnosis.

Data to examine includes the following:

CPU Usage

If there is a significant amount of idle CPU, then there could be an I/O, application, or database bottleneck. (Note that wait I/O should be considered as idle CPU).

If there is high CPU usage, then determine whether the CPU is being used effectively. Is the majority of CPU usage attributable to a small number of high-CPU using programs, or is the CPU consumed by an evenly distributed workload?

If the CPU is used by a small number of high-usage programs, then look at the programs to determine the cause.

Non-Oracle Processes If the programs are not Oracle programs, then identify whether they are legitimately requiring that amount of CPU. If so, then can their execution can be delayed to off-peak hours?

Oracle Processes If a small number of Oracle processes consumes most of the CPU resources, then use SQL_TRACE and TKPROF to identify the SQL or PL/SQL statements to see if a particular query or PL/SQL program unit can be tuned. For example, a SELECT statement could be CPU-intensive if it's execution involves many reads of data in cache (logical reads) that could be avoided with better SQL optimization.

Oracle CPU Statistics Oracle CPU statistics are available in three V$ views:

- 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. 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 when the Oracle Database Resource Manager is running.

Interpreting CPU Statistics It is important to recognize that CPU time and real time are distinct. With eight CPUs, for any given minute in real time, there are eight minutes of CPU time available. On NT and UNIX, this can be either user time or system time (*privileged* mode on NT). Thus, CPU time utilized by all processes (threads) on the system could be greater than one minute per one minute real time interval.

At any given moment, you know how much time Oracle has used on the system. So, if eight minutes are available and Oracle uses four 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. Identify the processes that are using CPU time, figure out why, and then attempt to tune them.

See Also: Chapter 10, "Using SQL Trace and TKPROF"

If the CPU usage is evenly distributed over many Oracle server processes, then examine the Statspack report for other evidence.

Detecting I/O Problems

An overly active I/O system can be evidenced by disk queue lengths greater than two, or disk service times that are over 20-30ms. If the I/O system is overly active, then check for potential hot spots that could benefit from distributing the I/O across more disks. Also identify whether the load can be reduced by lowering the resource requirements of the programs using those resources.

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 can also hold files that are not related to Oracle. 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$ views.

Tools, such as `sar -d` (or `iostat`) on many UNIX systems and Performance Monitor on Windows 2000 systems, examine I/O statistics for the entire system.

See Also: Your operating system documentation for the tools available on your platform

Check the Oracle wait event data in V$SYSTEM_EVENT to see whether the top wait events are I/O related. I/O related events include `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.

Cross reference the host I/O system data with the I/O sections in the Statspack report to identify hot datafiles and tablespaces. Also compare the I/O times reported by the OS with the times reported by Oracle to see if they are consistent.

Before investigating whether the I/O system should be reconfigured, determine if the load on the I/O system can be reduced. To reduce Oracle I/O load, look at SQL statements that perform many physical reads by querying the V$SQLAREA view or by reviewing the 'SQL ordered by physical reads' section of the Statspack report. Examine these statements to see how they can be tuned to reduce the number of I/Os.

If there are Oracle-related I/O problems caused by SQL statements, then tune them. If the Oracle server is not consuming the available I/O resources, then identify the process that is using up the I/O. Determine why the process is using up the I/O, and then tune this process.

See Also:

- Chapter 6, "Optimizing SQL Statements"
- "V$SQLAREA" on page 24-53
- Chapter 15, "I/O Configuration and Design"

Network

Using OS utilities, look at the network round-trip ping time and the number of collisions. If the network is causing large delays in response time, then investigate possible causes.

Network load can be reduced by scheduling large data transfers to off-peak times, or by coding applications to batch requests to remote hosts, rather than accessing remote hosts once (or more) per request.

See Also: *Oracle9i Database Performance Methods* for a description of important operating system statistics

Examine the Oracle Statistics

Oracle statistics are examined and cross-referenced with OS statistics to ensure a consistent diagnosis of the problem. OS statistics can indicate a good place to begin tuning. However, if the goal is to tune the Oracle instance, then look at the Oracle statistics to identify the resource bottleneck from Oracle's perspective before implementing corrective action.

See Also: "Interpreting Oracle Statistics" on page 22-12

Below are the common Oracle data sources used while tuning. The sources can be divided into two types of statistics: wait events and system statistics.

Wait Events

Wait events are statistics that are incremented by a server process/thread to indicate that it had to wait for an event to complete before being able to continue processing. Wait event data reveals various symptoms of problems that might be impacting

performance, such as latch contention, buffer contention, and I/O contention. Remember that these are only *symptoms* of problems—not the actual causes.

A server process can wait for the following:

- A resource to become available (such as a buffer or a latch)
- An action to complete (such as an I/O)
- More work to do (such as waiting for the client to provide the next SQL statement to execute). Events that identify that a server process is waiting for more work are known as *idle events*.

Wait event statistics include the number of times an event was waited for and the time waited for the event to complete. The views V$SESSION_WAIT, V$SESSION_EVENT, and V$SYSTEM_EVENT can be queried for wait event statistics. If the configuration parameter TIMED_STATISTICS is set to `true`, then you can also see how long each resource was waited for. To minimize user response time, reduce the time spent by server processes waiting for event completion. Not all wait events have the same wait time. Therefore, it is more important to examine events with the most total time waited rather than wait events with a high number of occurrences. Usually, it is best to set the dynamic parameter TIMED_STATISTICS to `true` at least while monitoring performance.

Investigate wait events and related timing data when performing reactive performance tuning. The events with the most time listed against them are often strong indications of the performance bottleneck. For example, by looking at V$SYSTEM_EVENT, you might notice lots of `buffer busy waits`. It might be that many processes are inserting into the same block and must wait for each other before they can insert. The solution could be to introduce freelists for the object in question.

System Statistics

System statistics are typically used in conjunction with wait event data to find further evidence of the cause of a performance problem.

For example, if V$SYSTEM_EVENT indicates that the largest wait event (in terms of wait time) is the event `buffer busy waits`, then look at the specific buffer wait statistics available in the view V$WAITSTAT to see which block type has the highest wait count and the highest wait time. After the block type has been identified, also look at V$SESSION_WAIT real-time while the problem is occurring to identify the contended-for object(s) using the file number and block number indicated. The combination of this data indicates the appropriate corrective action.

Statistics are available in many V$ views. Some common views include the following:

V$SYSSTAT This contains overall statistics for many different parts of Oracle, including rollback, logical and physical I/O, and parse data. Data from V$SYSSTAT is used to compute ratios, such as the buffer cache hit ratio.

V$FILESTAT This contains detailed file I/O statistics on a per-file basis, including the number of I/Os per file and the average read time.

V$ROLLSTAT This contains detailed rollback and undo segment statistics on a per-segment basis.

V$ENQUEUE_STAT This contains detailed enqueue statistics on a per-enqueue basis, including the number of times an enqueue was requested and the number of times an enqueue was waited for, and the wait time.

V$LATCH This contains detailed latch usage statistics on a per-latch basis, including the number of times each latch was requested and the number of times the latch was waited for.

> **See Also:** Chapter 24, "Dynamic Performance Views for Tuning" for detailed descriptions of the V$ views used in tuning

Implement and Measure Change

Often at the end of a tuning exercise, it is possible to identify two or three changes that could potentially alleviate the problem. To identify which change provides the most benefit, it is recommended that only one change be implemented at a time. The effect of the change should measured against the baseline data measurements found in the problem definition phase.

Typically, most sites with dire performance problems implement a number of overlapping changes at once, and thus cannot identify which changes provided any benefit. Although this is not immediately an issue, this becomes a significant hindrance if similar problems subsequently appear, because it is not possible to know which of the changes provided the most benefit and which efforts to prioritize.

If it is not possible to implement changes separately, then try to measure the effects of dissimilar changes. For example, measure the effect of making an initialization change to optimize redo generation separately from the effect of creating a new index to improve the performance of a modified query. It is *impossible* to measure

the benefit of performing an OS upgrade if SQL is tuned, the OS disk layout is changed, and the initialization parameters are also changed at the same time.

Performance tuning is an iterative process. It is unlikely to find a 'silver bullet' that solves an instance-wide performance problem. In most cases, excellent performance requires iteration through the performance tuning phases, because solving one bottleneck often uncovers another (sometimes worse) problem.

Knowing when to stop tuning is also important. The best measure of performance is user perception, rather than how close the statistic is to an ideal value.

Interpreting Oracle Statistics

Gather statistics that cover the time when the instance had the performance problem. If you previously captured baseline data for comparison, then you can compare the current data to the data from the baseline that most represents the problem workload.

When comparing two reports, ensure that the two reports are from times where the system was running comparable workloads.

See Also: "Principles of Data Gathering" on page 20-2

Examine Load

Usually, wait events are the first data examined. However, if you have a baseline report, then check to see if the load has changed. Regardless of whether you have a baseline, it is useful to see whether the resource usage rates are high.

Load-related statistics to examine include `redo size`, `session logical reads`, `db block changes`, `physical reads`, `physical writes`, `parse count (total)`, `parse count (hard)`, and `user calls`. This data is queried from V$SYSSTAT. It is best to normalize this data per second and per transaction.

In the Statspack report, look at the Load Profile section. (The data has been normalized per transaction and per second.)

Changing Load

The per second statistics show the changes in throughput (that is, whether the instance is performing more work per second). The per transaction statistics identify changes in the application characteristics by comparing these to the corresponding statistics from the baseline report.

High Rates of Activity

Examine the per second statistics to identify whether the 'rates' of activity are very high. It is difficult to make blanket recommendations on 'high' values, because the thresholds are different on each site and are contingent on the application characteristics, the number and speed of CPUs, the operating system, the I/O system, and the Oracle release.

Below are some generalized examples (acceptable values vary at each site):

- A hard parse rate of more than 100 per second indicates that there is a very high amount of hard parsing on the system. High hard parse rates cause serious performance issues and must be investigated. Usually, a high hard parse rate is accompanied by latch contention on the shared pool and library cache latches. Check whether waits for 'latch free' appear in the top-5 wait events, and if so, examine the latching sections of the Statspack report.
- A high soft parse rate could be in the rate of 300 or more per second. Unnecessary soft parses also limit application scalability. Optimally, a SQL statement should be soft parsed once per session and executed many times.

Using Wait Event Statistics to Drill Down to Bottlenecks

Whenever an Oracle process waits for something, it records the 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 are termed *idle* events, because the process is idle, waiting for work to perform. Nonidle events indicate nonproductive time spent waiting for a resource or action to complete.

Note: Not all symptoms can be evidenced by wait events. See "Additional Statistics" on page 22-16 for the statistics that can be checked.

The most effective way to use wait event data is to order the events by the wait time. This is only possible if `TIMED_STATISTICS` is set to `true`. Otherwise, the wait events can only be ranked by the number of times waited, which is often not the ordering that best represents the problem.

To get an indication of where time is spent, follow these steps:

1. Examine the data collection for `V$SYSTEM_EVENT`. The events of interest should be ranked by wait time.

Identify the wait events that have the most significant percentage of wait time. To determine the percentage of wait time, add the total wait time for all wait events, excluding idle events (such as `Null event`, `SQL*Net message from client`, `SQL*Net message to client`, `SQL*Net more data`). Calculate the relative percentage of the five most prominent events by dividing each event's wait time by the total time waited for all events

> **See Also:** "Idle Wait Events" for the complete list of idle events on page 22-43

Alternatively, look at the Top 5 Wait Events section on the front page of the Statspack report; this section automatically orders the wait events (omitting idle events), and calculates the relative percentage:

```
Top 5 Wait Events
~~~~~~~~~~~~~~~~~                                             Wait     % Total
Event                                               Waits   Time (cs)  Wt Time
----------------------------------------- ------------ ------------ -------
latch free                                      217,224       65,056   63.55
db file sequential read                          39,836       31,844   31.11
db file scattered read                            3,679        2,846    2.78
SQL*Net message from dblink                       1,186          870     .85
log file sync                                       830          775     .76
```

In the example above, the highest ranking wait event is the `latch free` event. In some situations, there might be a few events with similar percentages. This can provide extra evidence if all the events all related to the same type of resource request (for example, all I/O related events).

2. Look at the number of waits for these events, and the average wait time. For example, for I/O related events, the average time might help identify whether the I/O system is slow. Below is an example of this data taken from the Wait Event section of the Statspack report:

```
                                                          Total Wait   Avg wait  Waits
Event                                Waits   Timeouts  Time (s)    (ms)   /txn
-------------------------- ------------ ---------- ----------- ----- ------
latch free                    5,560,989  2,705,969      26,117     5  827.7
db file sequential read         137,027          0       2,129    16   20.4
SQL*Net break/reset to cli        1,566          0       1,707  1091    0.2
```

3. The top wait events identify the next places to investigate. A table of common wait events is listed in "Table of Wait Events and Potential Causes" below. For

the example above, the appropriate data to check would be latch-related. (It is usually a good idea to also have quick look at high-load SQL).

4. Examine the related data indicated by the wait events to see what other information this data provides. Determine whether this information is consistent with the wait event data. In most situations, there is enough data to begin developing a theory about the potential causes of the performance bottleneck.
5. To determine whether this theory is valid, cross-check data you have already examined with other statistics available for consistency. (The appropriate statistics vary depending on the problem, but usually include load profile-related data in V$SYSSTAT, OS statistics, and so on). Perform cross-checks with other data to confirm or refute the developing theory.

Table of Wait Events and Potential Causes

The table below links wait events to possible causes and gives an overview of the Oracle data that could be most useful to review next.

See Also:

- "Wait Events" on page 22-19 for detailed information on each event listed below and for other information to cross-check
- "Dynamic Performance Views for Tuning" for detailed information on querying V$ views

Table 22–1 Wait Events and Potential Causes

Wait Event	General Area	Possible Causes	Look For / Examine
buffer busy waits	Buffer cache, DBWR	Dependent on type of buffer: ■ index block in a primary key that is based on an ascending sequence ■ rollback segment header	Examine V$SESSION_WAIT while the problem is occurring to determine the type of block contended for.
free buffer waits	Buffer cache, DBWR, I/O	Slow DBWR (possibly due to I/O?) Cache too small	Examine write time using OS statistics. Check buffer cache statistics for evidence of too small cache.

Wait Event	General Area	Possible Causes	Look For / Examine
db file scattered read	I/O, SQL statement tuning	Poorly tuned SQL Slow I/O system	Investigate V$SQLAREA to see whether there are SQL statements performing many disk reads. Cross-check I/O system and V$FILESTAT for poor read time.
db file sequential read	I/O, SQL statement tuning	Poorly tuned SQL Slow I/O system	Investigate V$SQLAREA to see whether there are SQL statements performing many disk reads. Cross-check I/O system and V$FILESTAT for poor read time.
enqueue	Locks	Depends on type of enqueue	Look at V$ENQUEUE_STAT.
latch free	Latch contention	Depends on latch	Check V$LATCH.
log buffer space	Log buffer, I/O	Log buffer small Slow I/O system	Check the statistic redo buffer allocation retries in V$SYSSTAT. Check configuring log buffer section in configuring memory chapter. Check the disks that house the online redo logs for resource contention.
log file sync	I/O, over-committing	Slow disks that store the online logs Un-batched commits	Check the disks that house the online redo logs for resource contention. Check the number of transactions (commits + rollbacks) per second, from V$SYSSTAT.

Additional Statistics

There are a number of statistics that can indicate performance problems that do not have corresponding wait events.

Redo Log Space Requests Statistic

The V$SYSSTAT statistic redo log space requests indicates how many times a server process had to wait for space *in the online redo log*, not for space in the redo

log buffer. A significant value for this statistic and the wait events should be used as an indication that checkpoints, DBWR, or archiver activity should be tuned, not LGWR. Increasing the size of log buffer does not help.

Read Consistency

Your system might 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 changes are happening, then the query might need to roll back those changes often, in order to obtain a read-consistent image of the table. Compare the following V$SYSSTAT statistics to determine whether this is happening:
  - `consistent changes` statistic indicates the number of times a database block has rollback entries applied to perform a consistent read on the block. Workloads that produce a great deal of `consistent changes` can consume a great deal of resources.
  - `consistent gets` statistic counts the number of logical reads in consistent mode.
- If there are few very, large rollback segments, then your system could be spending a lot of time rolling back the transaction table during delayed block cleanout in order to find out exactly which SCN a transaction was committed. The ratio of the following V$SYSSTAT statistics should be close to 1:

  ```
  ratio = transaction tables consistent reads undo records applied /
          transaction tables consistent read rollbacks
  ```

  A solution is to create more, smaller rollback segments, or to use automatic undo management.
- If there are insufficient rollback segments, then there is rollback segment (header or block) contention. Evidence of this problem is available by the following:
  - Comparing the number of WAITS to the number of GETS in V$ROLLSTAT; the proportion of WAITS to GETS should be small.
  - Examining V$WAITSTAT to see whether there are many WAITS for buffers of CLASS 'undo header'.

  A solution is to create more rollback segments.

Table Fetch by Continued Row

You can detect migrated or chained rows by checking the number of `table fetch continued row` statistic in V$SYSSTAT. A small number of chained rows (less than 1%) is unlikely to impact system performance. However, a large percentage of chained rows can affect performance. This might not be relevant for segments that store LOBs, because these by nature can exceed the size of a block.

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.

Migration and chaining are especially detrimental to performance with the following:

- 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 in 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.

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.

See Also:

- *Oracle9i Database Concepts* for more information on PCTUSED
- *Oracle9i Database Administrator's Guide* for information on reorganizing tables

Parse-related Statistics

The more your application parses, 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 so SQL cannot be shared, or the shared pool is poorly configured.

See Also: Chapter 14, "Memory Configuration and Use"

There are a number of statistics available to identify the extent of time spent parsing by Oracle. Query the parse related statistics from V$SYSSTAT. For example:

```
SELECT NAME, VALUE
  FROM V$SYSSTAT
 WHERE NAME IN (  'parse time cpu', 'parse time elapsed'
                , 'parse count (hard)', 'CPU used by this session' );
```

There are various ratios that can be computed to assist in determining whether parsing may be a problem:

- `parse time CPU / parse time elapsed` This ratio indicates how much of the CPU time spent parsing was due to the parse operation itself, rather than waiting for resources, such as latches. A ratio of one is good, indicating that the elapsed time was not spent waiting for highly contended resources.
- `parse time CPU / CPU used by this session` This ratio indicates how much of the total CPU used by Oracle server processes was spent on parse-related operations. A ratio closer to zero is good, indicating that the majority of CPU is not spent on parsing.

Wait Events

The views V$SESSION_WAIT, V$SESSION_EVENT and V$SYSTEM_EVENT provide information on what resources were waited for, and, if the configuration parameter TIMED_STATISTICS is set to `true`, how long each resource was waited for.

Investigate wait events and related timing data when performing reactive performance tuning. The events with the most time listed against them are often strong indications of the performance bottleneck.

The three views contain related, but different, views of the same data:

- V$SESSION_WAIT is a current state view. It lists either the event currently being waited for or the event last waited for on each session
- V$SESSION_EVENT lists the cumulative history of events waited for on each session. After a session exits, the wait event statistics for that session are removed from this view.
- V$SYSTEM_EVENT lists the events and times waited for by the whole instance (that is, all session wait events data rolled up) since instance startup.

Because V$SESSION_WAIT is a current state view, it also contains a finer-granularity of information than V$SESSION_EVENT or V$SYSTEM_EVENT. It includes additional identifying data for the current event in three parameter columns: P1, P2, and P3.

For example, V$SESSION_EVENT can show that session 124 (SID=124) had many waits on the db file scattered read event, but it does not show which file and block number. However, V$SESSION_WAIT shows the file number in P1, the block number read in P2, and the number of blocks read in P3 (P1 and P2 let you determine for which segments the wait event is occurring).

This chapter concentrates on examples using V$SESSION_WAIT. However, Oracle recommends capturing performance data over an interval and keeping this data for performance and capacity analysis. This form of rollup data is queried from the V$SYSTEM_EVENT view by tools such as Enterprise Manager Diagnostics Pack and Statspack.

Most commonly encountered events are described in this chapter, listed in case-sensitive alphabetical order. Other event-related data to examine is also included. The case used for each event name is that which appears in the V$SYSTEM_EVENT view.

> **See Also:** *Oracle9i Database Reference* for a complete list of wait events

SQL*Net

The following events signify that the database process is waiting for acknowledgment from a database link or a client process:

- SQL*Net break/reset to client
- SQL*Net break/reset to dblink
- SQL*Net message from client
- SQL*Net message from dblink
- SQL*Net message to client
- SQL*Net message to dblink
- SQL*Net more data from client
- SQL*Net more data from dblink
- SQL*Net more data to client
- SQL*Net more data to dblink

If these waits constitute a significant portion of the wait time on the system or for a user experiencing response time issues, then the network or the middle-tier could be a bottleneck.

Events that are client-related should be diagnosed as described for the event SQL*Net message from client. Events that are dblink-related should be diagnosed as described for the event SQL*Net message from dblink.

SQL*Net message from client

Although this is an idle event, it is important to explain when this event can be used to diagnose what is *not* the problem. When a server process is waiting for work from the client process, it waits on this event. However, there are several situations where this event could accrue most of the wait time for a user experiencing poor response time.

Network Bottleneck This could be the case if the application causes a lot of traffic between the server and the client, and the network latency (time for a round-trip) is high. Symptoms include the following:

- Large number of waits for this event
- Both the database and client process are idle (waiting for network traffic) most of the time

To alleviate network bottlenecks, try the following:

- Tune the application to reduce round trips.

- Explore options to reduce latency (for example, terrestrial lines opposed to VSAT links).
- Change system configuration to move higher traffic components to lower latency links.

See Also: *Oracle9i Database Performance Methods*

Resource Bottleneck on the Client Process If the client process is using most of the resources, then there is nothing that can be done in the database. Symptoms include the following:

- Number of waits might not be large, but the time waited might be significant.
- Client process has a high resource usage.

In some cases, you can see the wait time for a waiting user tracking closely with the amount of CPU used by the client process. The term client here refers to any process other than the database process (middle-tier, desktop client) in the n-tier architecture.

SQL*Net Message from dblink

This event signifies that the session has sent a query to the remote node and is waiting for a response from the database link. This time could go up because of the following:

1. Network bottleneck
2. Time taken to run the query on the remote node

See Also:

- For #1, see "SQL*Net message from client" above.
- For #2, it is useful to see the query being run on the remote node. (Login to the remote database, find the session created by the database link, and examine the SQL statement being run by it).

buffer busy waits

This wait indicates that there are some buffers in the buffer cache that multiple processes are attempting to access concurrently. Query `V$WAITSTAT` for the wait statistics for each class of buffer. Common buffer classes that have buffer busy waits include `data block`, `segment header`, `undo header`, and `undo block`.

V$SESSION_WAIT Parameter Columns

P1 - File ID

P2 - Block ID

Causes

To determine the possible causes, identify the type of class contended for by querying V$WAITSTAT:

```
SELECT class, count
  FROM V$WAITSTAT
 WHERE count > 0
  ORDER BY count DESC;
```

Example output:

```
CLASS                   COUNT
------------------ ----------
data block              43383
undo header             10680
undo block               5237
segment header            785
```

To identify the segment and segment type contended for, query DBA_EXTENTS using the values for File Id and Block Id returned from V$SESSION_WAIT (p1 and p2 columns):

```
SELECT segment_owner, segment_name
  FROM DBA_EXTENTS
 WHERE file_id = <&p1>
   AND <&p2> BETWEEN block_id AND block_id + blocks - 1;
```

Actions

The action required depends on the class of block contended for and the actual segment.

segment header If the contention is on the segment header, then this is most likely freelist contention.

Automatic segment-space management in locally managed tablespaces eliminates the need to specify the PCTUSED, FREELISTS, and FREELIST GROUPS parameters. If possible, switch from manual space management to automatic segment-space management.

The following information is relevant if you are *unable* to use automatic segment-space management (for example, because the tablespace uses dictionary space management).

A freelist is a list of free data blocks that usually includes blocks existing in a number of different extents within the segment. Blocks in freelists 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 freelists for a database object must satisfy the PCTFREE and PCTUSED constraints. Specify the number of process freelists with the FREELISTS parameter. The default value of FREELISTS is one. The maximum value depends on the data block size.

To find the current setting for freelists for that segment, run the following:

```
SELECT SEGMENT_NAME, FREELISTS
  FROM DBA_SEGMENTS
 WHERE SEGMENT_NAME = <segment name>
   AND SEGMENT_TYPE = <segment type>;
```

Set freelists, or increase of number of freelists. If adding more freelists does not alleviate the problem, then use freelist groups (even in single instance this can make a difference). If using Oracle Real Application Clusters, then ensure that each instance has its own freelist group(s).

See Also:

- *Oracle9i Database Concepts* for information on automatic segment-space management, freelists, PCTFREE, and PCTUSED
- *Oracle9i Real Application Clusters Installation and Configuration* for information about using freelist groups in an Oracle Real Application Clusters environment

data block If the contention is on tables or indexes (not the segment header):

- Check for SQL statements using unselective indexes
- Check for 'right-hand-indexes' (that is, indexes that are inserted into at the same point by many processes; for example, those which use sequence number generators for the key values)
- Consider using automatic segment-space management, or increasing freelists to avoid multiple processes attempting to insert into the same block

undo header For contention on rollback segment header:

- If you are not using automatic undo management, then add more rollback segments

undo block For contention on rollback segment block:

- If you are not using automatic undo management, then consider making rollback segment sizes larger

db file scattered read

This event signifies that the user process is reading buffers into the SGA buffer cache and is waiting for a physical I/O call to return. A `db file scattered read` issues a scatter-read to read the data into multiple discontinuous memory locations. A scattered read is usually a multiblock read.

The `db file scattered read` wait event identifies that a full table scan is occurring. When performing a full table scan into the buffer cache, the blocks read are read into memory locations that are not physically adjacent to each other. Such reads are called scattered read calls, because the blocks are scattered throughout memory. This is why the corresponding wait event is called 'db file scattered read'. Multiblock (up to `DB_FILE_MULTIBLOCK_READ_COUNT` blocks) reads due to full table scans into the buffer cache show up as waits for 'db file scattered read'.

V$SESSION_WAIT Parameter Columns

`P1` - The absolute file number

`P2` - The block being read

`P3` - The number of blocks (should be greater than 1)

Actions

On a healthy system, physical read waits should be the biggest waits after the idle waits. However, also consider the following:

- Are there direct read waits (signifying full table scans with parallel query) or `db file scattered read` waits on an operational (OLTP) system that should be doing small indexed accesses?
- Are there `db file sequential reads` on a large data warehouse that should be seeing mostly full table scans with parallel query?

Other things that could indicate excessive I/O load on the system include the following:

- Poor buffer cache hit ratio
- These events accruing most of the wait time for a user experiencing poor response time

Managing Excessive I/O

There are several ways to handle excessive I/O waits. In the order of effectiveness, these are as follows:

1. Reduce the I/O activity by SQL tuning.
2. Reduce the need to do I/O by managing the workload.
3. Add more disks to reduce the number of I/Os per disk.
4. Alleviate I/O hot spots by redistributing I/O across existing disks.

See Also: "I/O Configuration and Design"

The first course of action should be to find opportunities to reduce I/O. Examine the SQL statements being run by sessions waiting for these events, as well as statements causing high physical I/Os from V$SQLAREA. Factors that can adversely affect the execution plans causing excessive I/O include the following:

- Improperly optimized SQL
- Missing indexes
- High degree of parallelism for the table (skewing the optimizer towards scans)
- Lack of accurate statistics for the optimizer

Inadequate I/O Distribution

Besides reducing I/O, also examine the I/O distribution of files across the disks. Is I/O distributed uniformly across the disks, or are there hot spots on some disks? Are the number of disks is sufficient to meet the I/O needs of the database?

See the total I/O operations (reads and writes) by the database, and compare those with the number of disks used. Remember to include the I/O activity of LGWR and ARCH processes.

Finding the SQL Statement executed by Sessions Waiting for I/O

Use the following query to find the SQL statement for sessions waiting for I/O:

```
SELECT s.sql_hash_value
  FROM V$SESSION s, V$SESSION_WAIT w
 WHERE w.event LIKE 'db file%read'
   AND w.sid = s.sid ;
```

Finding the Object Requiring I/O

Use the following query to find the object being accessed:

```
SELECT segment_owner, segment_name
  FROM DBA_EXTENTS
 WHERE file_id = &p1
   AND &p2 between block_id AND block_id + blocks - 1 ;
```

db file sequential read

This event signifies that the user process is reading buffers into the SGA buffer cache and is waiting for a physical I/O call to return. This call differs from a scattered read, because a sequential read is reading data into contiguous memory space. A sequential read is usually a single-block read.

Single block I/Os are usually the result of using indexes. Rarely, full table scan calls could get truncated to a single block call due to extent boundaries, or buffers already present in the buffer cache. These waits would also show up as 'db file sequential read'.

V$SESSION_WAIT Parameter Columns

P1 - The absolute file number

P2 - The block being read

P3 - The number of blocks (should be 1)

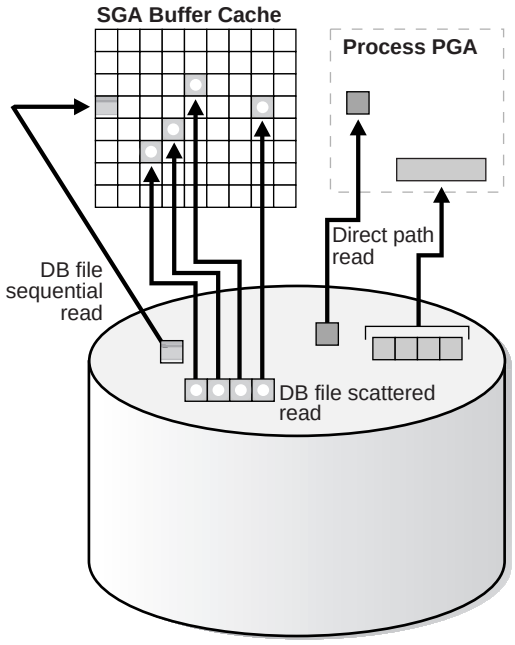
> **See Also:** "db file scattered read" on page 22-25 for information on managing excessive I/O, inadequate I/O distribution, and finding the SQL causing the I/O and the segment the I/O is performed on

Figure 22–1 depicts the differences between the following wait events:

- db file sequential read (single block read into one SGA buffer)
- db file scattered read (multiblock read into many discontinuous SGA buffers)

- `direct read` (single or multiblock read into the PGA, bypassing the SGA)

Figure 22–1 Scattered Read, Sequential Read, and Direct Path Read



direct path read

When a session is reading buffers from disk directly into the PGA (opposed to the buffer cache in SGA), it waits on this event. If the I/O subsystem does not support asynchronous I/Os, then each wait corresponds to a physical read request.

If the I/O subsystem supports asynchronous I/O, then the process is able to overlap issuing read requests with processing the blocks already existing in the PGA. When the process attempts to access a block in the PGA that has not yet been read from disk, it then issues a wait call and updates the statistics for this event. Hence, the number of waits is not necessarily the same as the number of read requests (unlike 'db file scattered read' and 'db files sequential read').

V$SESSION_WAIT Parameter Columns

P1 - File_id for the read call

P2 - Start block_id for the read call

P3 - Number of blocks in the read call

Causes

This happens in the following situations:

- The sorts are too large to fit in memory and go to disk.
- Parallel slaves are used for scanning data.
- The server process is processing buffers faster than the I/O system can return the buffers. This can indicate an overloaded I/O system.

Actions

The `file_id` shows if the reads are for an object in `TEMP` tablespace (sorts to disk) or full table scans by parallel slaves. This is the biggest wait for large data warehouse sites. However, if the workload is not a DSS workload, then examine why this is happening.

Sorts to Disk Examine the SQL statement currently being run by the session experiencing waits to see what is causing the sorts. Query `V$SORT_USAGE` to find the SQL statement that is generating the sort. Also query the statistics from `V$SESSTAT` for the session to determine the size of the sort. See if it is possible to reduce the sorting by tuning the SQL statement. If `WORKAREA_SIZE_POLICY` is `MANUAL`, then consider increasing the `SORT_AREA_SIZE` for the system (if the sorts are not too big) or for individual processes. If `WORKAREA_SIZE_POLICY` is `AUTO`, then investigate whether to increase `PGA_AGGREGATE_TARGET`.

See Also: "Configuring the PGA Working Memory" on page 14-46

Full Table Scans If tables are defined with a high degree of parallelism, then this could skew the optimizer to use full table scans with parallel slaves. Check the object being read into using the direct path reads, as well as the SQL statement being run by the query-coordinator. If the full table scans are a valid part of the workload, then ensure that the I/O subsystem is sized adequately for the degree of parallelism.

Hash Area Size For query plans that call for a hash join, excessive I/O could result from having HASH_AREA_SIZE too small. If WORKAREA_SIZE_POLICY is MANUAL, then consider increasing the HASH_AREA_SIZE for the system or for individual processes. If WORKAREA_SIZE_POLICY is AUTO, then investigate whether to increase PGA_AGGREGATE_TARGET.

See Also:

- "Managing Excessive I/O" on page 22-26
- "Configuring the PGA Working Memory" on page 14-46

direct path write

When a process is writing buffers directly from PGA (as opposed to the DBWR writing them from the buffer cache), the process waits on this event for the write call to complete. Operations that could perform direct path writes include when a sort goes to disk, during parallel DML operations, direct-path INSERTs, parallel create table as select, and some LOB operations.

Like direct path reads, the number of waits is not the same as number of write calls issued if the I/O subsystem supports asynchronous writes. The session waits if it has processed all buffers in the PGA and is unable to continue work until an I/O request completes.

V$SESSION_WAIT Parameter Columns

P1 - File_id for the write call

P2 - Start block_id for the write call

P3 - Number of blocks in the write call

Causes

This happen in the following situations:

- Sorts are too large to fit in memory and are going to disk
- Parallel DML are issued to create/populate objects

Actions

For large sorts see "Sorts to Disk" on page 22-29.

For parallel DML, check the I/O distribution across disks and make sure that the I/O subsystem is adequately sized for the degree of parallelism.

enqueue

Enqueues are locks that serialize access to database resources. This event indicates that the session is waiting for a lock that is held by another session.

V$SESSION_WAIT Parameter Columns

P1 - Name and type of the lock

P2 - Resource identifier ID1 for the lock

P3 - Resource identifier ID2 for the lock

Comparison with V$LOCK Columns

V$LOCK.ID1 = P2

V$LOCK.ID2 = P3

Performing the following SQL transformation of the P1 column results in the same value displayed in V$LOCK.TYPE:

```
V$LOCK.TYPE = chr(bitand(P1,-16777216)/16777215)||
chr(bitand(P1,16711680)/65535)
```

Actions

The appropriate action depends on the type of enqueue.

ST enqueue If the contended-for enqueue is the ST enqueue, then the problem is most likely dynamic space allocation. Oracle dynamically allocates an extent to a segment when there is no more free space available in the segment. This enqueue is only used for dictionary managed tablespaces.

To solve contention on this resource:

- Check to see whether the temporary (that is, sort) tablespace uses TEMPFILES. If not, then switch to using TEMPFILES.
- Switch to using locally managed tablespaces if the tablespace that contains segments that are growing dynamically.

  See Also: *Oracle9i Database Concepts* for detailed information on TEMPFILEs and locally managed tablespaces

- If it is not possible to switch to locally managed tablespaces, then ST enqueue resource usage can be decreased by changing the next extent sizes of the

growing objects to be large enough to avoid constant space allocation. To determine which segments are growing constantly, monitor the EXTENTS column of the DBA_SEGMENTS view for all SEGMENT_NAMEs over time to identify which segments are growing and how quickly.

- Preallocate space in the segment (for example, by allocating extents using the ALTER TABLE ALLOCATE EXTENT SQL statement).

Other Locks Query V$LOCK to find the sessions holding the lock. For every session waiting for the event enqueue, there is a row in V$LOCK with REQUEST <> 0. Therefore, use either of the two queries to find the sessions holding the locks and waiting for the locks.

```
SELECT DECODE(l.request,0,'Holder: ','Waiter: ')||sid sess
     , id1, id2, lmode, request, type
  FROM V$LOCK l
 WHERE (l.id1,l.id2,l.type) IN
      ( SELECT w.p2, w.p3,   chr(bitand(w.p1,-16777216)/16777215)
                           || chr(bitand(w.p1,16711680)/65535)
          FROM V$SESSION_WAIT w
         WHERE w.wait_time = 0 AND w.event = 'enqueue' )
ORDER BY l.id1, l.request;

SELECT DECODE(request,0,'Holder: ','Waiter: ')||sid sess
     , id1, id2, lmode, request, type
  FROM V$LOCK
 WHERE (id1, id2, type) IN
             (SELECT id1, id2, type FROM V$LOCK WHERE lmode = 0)
 ORDER BY id1, request;
```

See Also:

- "Dynamic Performance Views for Tuning" for more information on using V$LOCK
- *Oracle9i Database Reference* for more information on enqueues

Other Locks - HW enqueue The HW enqueue is used to serialize the allocation of space above the high-water mark of a segment.

V$SESSION_WAIT.P2 / V$LOCK.ID1 is the tablespace number

V$SESSION_WAIT.P2 / V$LOCK.ID2 is the relative dba of segment header of the object for which space is being allocated.

If this is a point of contention for an object, then manual allocation of extents solves the problem.

Other Locks - TM enqueue The most common reason for waits on TM locks tend to involve foreign key constraints where the constrained columns are not indexed. Index the foreign key columns to avoid this problem.

Other Locks - TX enqueue These are acquired exclusive when a transaction initiates its first change and held until the transaction does a `COMMIT` or `ROLLBACK`.

- TX in mode 6: occurs when a session is waiting for a row level lock that is already held by another session. This occurs when one user is updating (or deleting) a row, which another session wishes to update or delete.

  The solution is to have the first session already holding the lock perform a `COMMIT` or `ROLLBACK`.

- TX in mode 4 can occur if the session is waiting for an ITL (interested transaction list) slot in a block. This happens when the session wants to lock a row in the block but one or more other sessions have rows locked in the same block, and there is no free ITL slot in the block. Usually, Oracle dynamically adds another ITL slot. This may not be possible if there is insufficient free space in the block to add an ITL. If so, the session waits for a slot with a TX enqueue in mode 4.

  The solution is to increase the number of ITLs available, either by changing the `INITTRANS` or `MAXTRANS` for the table (either by using an `ALTER` statement, or by recreating the table with the higher values).

- TX in mode 4 can also occur if a session is waiting due to potential duplicates in `UNIQUE` index. If two sessions try to insert the same key value the second session has to wait to see if an `ORA-0001` should be raised or not.

  The solution is to have the first session already holding the lock perform a `COMMIT` or `ROLLBACK`.

- TX in mode 4 is also possible if the session is waiting due to shared bitmap index fragment. Bitmap indexes index key values and a range of ROWIDs. Each 'entry' in a bitmap index can cover many rows in the actual table. If two sessions want to update rows covered by the same bitmap index fragment, then the second session waits for the first transaction to either `COMMIT` or `ROLLBACK` by waiting for the TX lock in mode 4.

free buffer waits

This wait event indicates that a server process was unable to find a free buffer and has posted the database writer to make free buffers by writing out dirty buffers. A dirty buffer is a buffer whose contents have been modified. Dirty buffers are freed for reuse when DBWR has written the blocks to disk.

Causes

DBWR may not be keeping up with writing dirty buffers in the following situations:

- The I/O system is slow.
- There are resources it is waiting for, such as latches.
- The buffer cache is so small that DBWR spends most of it's time cleaning out buffers for server processes.
- The buffer cache is so big that one DBWR process is not enough to free enough buffers in the cache to satisfy requests.

Actions

If this event occurs frequently, then examine the session waits for DBWR to see whether there is anything delaying DBWR.

Writes If it is waiting for writes, then determine what is delaying the writes and fix it. Check the following:

- Examine `V$FILESTAT` to see where most of the writes are happening.
- Examine the host OS statistics for the I/O system. Are the write times acceptable?

If I/O is slow:

- Consider using faster I/O alternatives to speed up write times.
- Spread the I/O activity across large number of spindles (disks) and controllers.

> **See Also:** Chapter 15, "I/O Configuration and Design" for information on balancing I/O

Cache is Too Small It is possible DBWR is very active because of the cache is too small. Investigate whether this is a probable cause by looking to see if the buffer cache hit ratio is low. Also use the `V$DB_CACHE_ADVICE` view to determine whether a larger cache size would be advantageous.

See Also: "Sizing the Buffer Cache" on page 14-5

Cache Is Too Big for One DBWR If the cache size is adequate and the I/O is already evenly spread, then you can potentially modify the behavior of DBWR by using asynchronous I/O or by using multiple database writers.

Consider Multiple Database Writer (DBWR) Processes or I/O Slaves

Configuring multiple database writer processes, or using I/O slaves, is useful when the transaction rates are high or when the buffer cache size is so large that a single DBW*n* process cannot keep up with the load.

DB_WRITER_PROCESSES The `DB_WRITER_PROCESSES` initialization parameter lets you configure multiple database writer processes (from DBW0 to DBW9). Configuring multiple DBWR processes distributes the work required to identify buffers to be written, and it also distributes the I/O load over these processes.

DBWR_IO_SLAVES If it is not practical to use multiple DBWR processes, then Oracle provides a facility whereby the I/O load can be distributed over multiple slave processes. The DBWR process is the only process that scans the buffer cache LRU list for blocks to be written out. However, the I/O for those blocks is performed by the I/O slaves. The number of I/O slaves is determined by the parameter `DBWR_IO_SLAVES`. I/O slaves are also useful when asynchronous I/O is not available, because the multiple I/O slaves simulate nonblocking, asynchronous requests by freeing DBWR to continue identifying blocks in the cache to be written.

DBWR I/O slaves are allocated immediately following database open when the first I/O request is made. The DBWR continues to perform all of the DBWR-related work, apart from performing I/O. I/O slaves simply perform the I/O on behalf of DBWR. The writing of the batch is parallelized between the I/O slaves.

Note: Implementing `DBWR_IO_SLAVES` requires that extra shared memory be allocated for I/O buffers and request queues. Multiple DBWR processes cannot be used with I/O slaves. Configuring I/O slaves forces only one DBWR process to start.

Choosing Between Multiple DBWR Processes and I/O Slaves Configuring multiple DBWR processes benefits performance when a single DBWR process is unable to keep up with the required workload. However, before configuring multiple DBWR processes, check whether asynchronous I/O is available and configured on the

system. If the system supports asynchronous I/O but it is not currently used, then enable asynchronous I/O to see if this alleviates the problem. If the system does not support asynchronous I/O, or if asynchronous I/O is already configured and there is still a DBWR bottleneck, then configure multiple DBWR processes.

> **Note:** If asynchronous I/O is not available on your platform, then asynchronous I/O can be disabled by setting the `DISK_ASYNCH_IO` initialization parameter to `false`.

Using multiple DBWRs parallelizes the gathering and writing of buffers. Therefore, multiple DBW*n* processes should deliver more throughput than one DBWR process with the same number of I/O slaves. For this reason, the use of I/O slaves has been deprecated in favor of multiple DBWR processes. I/O slaves should only be used if multiple DBWR processes cannot be configured.

> **See Also:** Chapter 17, "Configuring Instance Recovery Performance" for details on tuning checkpoints

latch free

A latch is a low-level internal lock used by Oracle to protect memory structures. The latch free event is updated when a server process attempts to get a latch, and the latch is unavailable on the first attempt.

> **See Also:** *Oracle9i Database Concepts* for more information on latches and internal locks

Actions

This event should only be a concern if latch waits are a significant portion of the wait time on the system as a whole, or for individual users experiencing problems.

- To help determine the cause of this wait event, identify the latch(es) contended for. There are many types of latches used for different purposes. For example, the shared pool latch protects certain actions in the shared pool, and the cache buffers LRU chain protects certain actions in the buffer cache.
- Examine the resource usage for related resources. For example, if the library cache latch is heavily contended for, then examine the hard and soft parse rates.
- Examine the SQL statements for the sessions experiencing latch contention to see if there is any commonality.

V$SESSION_WAIT Parameter Columns

P1 - Address of the latch

P2 - Latch number

P3 - Number of times process has already slept, waiting for the latch

Example: Find Latches Currently Waiting For

```
SELECT n.name, SUM(w.p3) Sleeps
  FROM V$SESSION_WAIT w, V$LATCHNAME n
 WHERE w.event = 'latch free'
   AND w.p2 = n.latch#
 GROUP BY n.name;
```

Table 22–2 Latch Free Wait Event

Latch	SGA Area	Possible Causes	Look For:
Shared pool, library cache	Shared pool	Lack of statement reuse Statements not using bind variables Insufficient size of application cursor cache Cursors closed explicitly after each execution Frequent logon / logoffs Underlying object structure being modified (for example truncate) Shared pool too small	Sessions (in V$SESSTAT) with high: ■ parse time CPU ■ parse time elapsed ■ Ratio of parse count (hard) / execute count ■ Ratio of parse count (total) / execute count Cursors (in V$SQLAREA / V$SQL) with: ■ High ratio of PARSE_CALLS / EXECUTIONS ■ EXECUTIONS = 1 differing only in literals in the WHERE clause (that is, no bind variables used) ■ High RELOADS ■ High INVALIDATIONS ■ Large (> 1mb) SHARABLE_MEM

Table 22–2 Latch Free Wait Event

Latch	SGA Area	Possible Causes	Look For:
cache buffers lru chain	Buffer cache LRU lists	Excessive buffer cache throughput. For example, many cache-based sorts, inefficient SQL that accesses incorrect indexes iteratively (large index range scans), or many full table scans	Statements with very high LIO / PIO using unselective indexes
		DBWR not keeping up with the dirty workload; hence, foreground process spends longer holding the latch looking for a free buffer	
		Cache may be too small	
cache buffers chains	Buffer cache buffers	Repeated access to a block (or small number of blocks), known as 'hot block'	Sequence number generation code that updates a row in a table to generate the number, rather than using a sequence number generator
			Identify the segment the hot block belongs to

Shared Pool and Library Cache Latch Contention

A main cause of shared pool or library cache latch contention is parsing. There are a number of techniques that can be used to identify unnecessary parsing and a number of types of unnecessary parsing:

Unshared SQL This method identifies similar SQL statements that could be shared if literals were replaced with bind variables. The idea is to either:

- Manually inspect SQL statements that have only one execution to see whether they are similar:

```
SELECT sql_text
  FROM V$SQLAREA
 WHERE executions = 1
 ORDER BY sql_text;
```

- Or, automate this process by grouping together what may be similar statements. Do this by estimating the number of bytes of a SQL statement which will likely

be the same, and group the SQL statements by that many bytes. For example, the example below groups together statements that differ only after the first 60 bytes.

```
SELECT SUBSTR(sql_text,1, 60), COUNT(*)
  FROM V$SQLAREA
 WHERE executions = 1
 GROUP BY SUBSTR(sql_text, 1, 60)
 HAVING COUNT(*) > 1;
```

Reparsed Sharable SQL check the V$SQLAREA view. Enter the following query:

```
SELECT SQL_TEXT, PARSE_CALLS, EXECUTIONS
FROM V$SQLAREA
ORDER BY PARSE_CALLS;
```

When the PARSE_CALLS value is close to the EXECUTIONS value for a given statement, you might be continually reparsing that statement. Tune the statements with the higher numbers of parse calls.

By Session Identify unnecessary parse calls by identifying the session in which they occur. It might be that particular batch programs or certain types of applications do most of the reparsing. To do this, run the following query:

```
column sid format 99999
column name format a20
SELECT ss.sid, sn.name, ss.value
  FROM V$SESSTAT  ss
     , V$STATNAME sn
 WHERE name IN ('parse count (hard)','execute count')
   AND ss.statistic# = sn.statistic#
   AND ss.value > 0
 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. The output is similar to the following:

```
   SID NAME                     VALUE
------ -------------------- ----------
     7 parse count (hard)            1
     8 parse count (hard)            3
     7 execute count                20
     6 parse count (hard)           26
    11 parse count (hard)           84
     6 execute count               325
    11 execute count              1619
     8 execute count             12690
```

cache buffer lru chain The cache buffer lru chain latches protect the lists of buffers in the cache. When adding, moving, or removing a buffer from a list, a latch must be obtained.

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. LRU latch contention is detected by querying V$LATCH, V$SESSION_EVENT, and V$SYSTEM_EVENT. To avoid contention, consider bypassing the buffer cache or redesigning the application.

cache buffers chains The cache buffers chains latches are used to protect a buffer list in the buffer cache. These latches are used when searching for, adding, or removing a buffer from the buffer cache. Contention on this latch usually means that there is a block that is greatly contended for (that is, 'hot)' block.

To identify the heavily accessed buffer chain, and hence the contended for block, look at latch statistics for the cache buffers chains latches using the view V$LATCH_CHILDREN. If there is a specific cache buffers chains child latch that has many more GETS, MISSES, and SLEEPS when compared with the other child latches, then this is the contended for child latch.

This latch has a memory address, identified by the ADDR column. Use the value in the ADDR column joined with the V$BH view to identify the blocks protected by this latch. For example, given the address (V$LATCH_CHILDREN.ADDR) of a heavily contended latch, this queries the file and block numbers:

```
SELECT file#, dbablk, class, state
  FROM X$BH
 WHERE HLADDR='address of latch';
```

There are many blocks protected by each latch. One of these buffers will likely be the hot block. Perform this query a number of times, and identify the block that consistently appears in the output, using the combination of file number (`file#`) and block number (`dbablk`). This is most likely the hot block. After the hot block has been identified, query `DBA_EXTENTS` using the file number and block number, to identify the segment.

> **See Also:** "Finding the Object Requiring I/O" on page 22-27 for instructions on how to do this

log buffer space

This event occurs when server processes are waiting for free space in the log buffer, because you are writing redo to the log buffer faster than LGWR can write it out.

Actions

Modify the redo log buffer size. If the size of the log buffer is already reasonable, then ensure that the disks on which the online redo logs reside do not suffer from I/O contention. The `log buffer space` wait event could be indicative of either disk I/O contention on the disks where the redo logs reside, or of a too-small log buffer. Check the I/O profile of the disks containing the redo logs to investigate whether the I/O system is the bottleneck. If the I/O system is not a problem, then the redo log buffer could be too small. Increase the size of the redo log buffer until this event is no longer significant.

log file switch

There are two wait events commonly encountered:

- `log file switch (archiving needed)`
- `log file switch (checkpoint incomplete)`

In both of the events, the LGWR is unable to switch into the next online redo log, and all the commit requests wait for this event.

Actions

For the `log file switch (archiving needed)` event, examine why the archiver is unable to archive the logs in a timely fashion. It could be due to the following:

- Archive destination is running out of free space.
- Archiver is not able to read redo logs fast enough (contention with the LGWR).

- Archiver is not able to write fast enough (contention on the archive destination, or not enough ARCH processes).

Depending on the nature of bottleneck, you might need to redistribute I/O or add more space to the archive destination to alleviate the problem. For the `log file switch (checkpoint incomplete)` event:

- Check if DBWR is slow, possibly due to an overloaded or slow I/O system. Check the DBWR write times, check the I/O system, and distribute I/O if necessary.

  See Also: Chapter 15, "I/O Configuration and Design"

- Check if there are too few, or too small redo logs. If you have a few and/or small redo logs (for example two x 100k logs), and your system produces enough redo to cycle through all of the logs before DBWR has been able to complete the checkpoint, then increase the size and/or number of redo logs.

  See Also: "Sizing Redo Log Files" on page 13-5

log file sync

When a user session commits (or rolls back), the session's redo information must be flushed to the redo logfile by LGWR. The server process performing the `COMMIT` or `ROLLBACK` waits under this event for the write to the redo log to complete.

Actions

If this event's waits constitute a significant wait on the system or a significant amount of time waited by a user experiencing response time issues or on a system, then examine the time per wait.

If the average time waited is low, but the number of waits are high, then the application might be committing after every `INSERT`, rather than batching `COMMIT`s. Applications can reduce the wait by committing after 50 rows, rather than every row.

If the time per wait is high, then examine the session waits for the log writer and see what it is spending most of it's time doing and waiting for. If the waits are because of slow I/O, then try the following:

- Reduce other I/O activity on the disks containing the redo logs, or use dedicated disks.
- Alternate redo logs on different disks to minimize the effect of the archiver on the log writer.
- Move the redo logs to faster disks or a faster I/O subsystem (for example, switch from RAID 5 to RAID 1).
- Consider using raw devices (or simulated raw devices provided by disk vendors) to speed up the writes.
- Depending on the type of application, it might be possible to batch `COMMIT`s by committing every *N* rows, rather than every row, so that fewer log file syncs are needed.

rdbms ipc reply

This event is used to wait for a reply from one of the background processes.

Idle Wait Events

These events indicate that the server process is waiting because it has no work. This usually implies that if there is a bottleneck, then the bottleneck is not for database resources.

The majority of the idle events should be ignored when tuning, because they do not indicate the nature of the performance bottleneck. Some idle events can be useful in indicating what the bottleneck *is not*. An example of this type of event is the most commonly encountered idle wait-event 'SQL Net message from client'. This and other idle events (and their categories) are listed below.

Table 22–3 Idle Wait Events

Wait Name	Background Process Idle Event	User Process Idle Event	Parallel Query Idle Event	Shared Server Idle Event	Oracle Real Application Clusters Idle Event
`dispatcher timer`				X	
`lock manager wait for remote message`					X
`pipe get`		X			

Table 22–3 Idle Wait Events

Wait Name	Background Process Idle Event	User Process Idle Event	Parallel Query Idle Event	Shared Server Idle Event	Oracle Real Application Clusters Idle Event
pmon timer	X				
PX Idle Wait			X		
PX Deq Credit: need buffer			X		
PX Deq Credit: send blkd			X		
rdbms ipc message	X				
smon timer	X				
SQL*Net message from client		X			
virtual circuit status				X	

Note: If Statspack is installed, then it is also possible to query the STATS$IDLE_EVENT table, which contains a list of idle events.

See Also: *Oracle9i Database Reference* for explanations of each idle wait event

23

Tuning Networks

This chapter describes different connection models and 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. You can have a shared server configuration or a dedicated server configuration. If you have a shared server configuration, then LSNRCTL services lists dispatchers. If you have a dedicated server configuration, then LSNRCTL services lists dedicated servers.

It is possible to connect to dedicated server with a database configured for shared servers by placing the parameter (SERVER = DEDICATED) in the connect descriptor.

Shared Server 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 dispatchers have been started successfully.

> **Note:** Remember that PMON can take a minute to register the dispatcher with the listener.

```
LSNRCTL> services
Connecting to
(DESCRIPTION=(ADDRESS=(PROTOCOL=tcp)(HOST=helios)(PORT=1521)))
Services Summary...
Service "sales.us.acme.com" has 1 instance(s).
  Instance "sales", status READY, has 3 handler(s) for this service...
    Handler(s):
      "DEDICATED" established:0 refused:0 state:ready
         LOCAL SERVER
      "D000" established:0 refused:0 current:0 max:10000 state:ready
         DISPATCHER <machine: helios, pid: 1689>
         (ADDRESS=(PROTOCOL=tcp)(HOST=helios)(PORT=52414))
      "D001" established:0 refused:0 current:0 max:10000 state:ready
         DISPATCHER <machine: helios, pid: 1691>
         (ADDRESS=(PROTOCOL=tcp)(HOST=helios)(PORT=52415))
The command completed successfully.
```

> **See Also:** *Oracle Net Services Administrator's Guide* for information on setting the output mode

Configuring the Initialization Parameter File

- Make sure that the DISPATCHERS line is correctly set. For example:

```
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 previous example, the entire DISPATCHERS line can be "(PROTOCOL=TCP)". The attributes DISPATCHERS, LISTENER, SERVICE, SESSIONS, CONNECTIONS, MULTIPLEX, POOL, and TICKS are all optional.

  > **See Also:** *Oracle9i Database Reference* and *Oracle Net Services Administrator's Guide* for more information on these parameters

- Make sure that the optional MAX_DISPATCHERS line is correctly set. For example:

```
MAX_DISPATCHERS = 4
```

  This line should reflect the total number of dispatchers you want to start.

- Make sure that the optional MAX_SHARED_SERVERS line is correctly set. For example:

```
MAX_SHARED_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 MAX_SHARED_SERVERS to more then this.

```
SELECT maximum_connections "MAX CONN", servers_started "STARTED",
    servers_terminated "TERMINATED", servers_highwater "HIGHWATER"
FROM V$SHARED_SERVER_MONITOR;
```

- Make sure that the optional SHARED_SERVERS line is correctly set. For example:

  ```
  SHARED_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 be used when the database is active. MAX_SHARED_SERVERS is intended to handle peak load.

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> services
Connecting to
(DESCRIPTION=(ADDRESS=(PROTOCOL=tcp)(HOST=helios)(PORT=1521)))
Services Summary...
Service "sales.us.acme.com" has 1 instance(s).
  Instance "sales", status READY, has 2 handler(s) for this service...
    Handler(s):
      "DEDICATED" established:11 refused:0 state:ready
         LOCAL SERVER
      "D000" established:565 refused:4 current:155 max:10000 state:ready
         DISPATCHER <machine: helios, pid: 5673>
         (ADDRESS=(PROTOCOL=tcp)(HOST=helios)(PORT=38411))
The command completed successfully.
```

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: *Oracle Net Services Administrator's Guide*

Stop and restart the listener to initialize this new parameter. If you are not currently running a shared server configuration, then consider doing so. It is faster for the listener to handle a client request in a shared server configuration than it is in a dedicated server configuration.

Note: Shared server dispatchers also receive connect requests and can also benefit from tuning the queue size.

The maximum queue size is subject to the maximum size possible for a particular operating system.

Detecting Network Problems

This section encompasses local area network (LAN) and wide area network (WAN) troubleshooting methods.

Using Dynamic Performance Views for Network Performance

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.

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 23–1 lists the most common ratings for various topologies.

Table 23–1 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

Table 23–1 Bandwidth Ratings

Topology or Carrier	Bandwidth
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. Usually, it is used with slower modems.
Dial Up Modems	56 Kilobits / second. Usually, it is accompanied with data compression for faster throughput.

Solving Network Problems

This section describes several techniques for enhancing performance and solving network problems.

- Finding Network Bottlenecks
- Dissecting Network Bottlenecks
- Using Array Interfaces
- Adjusting Session Data Unit Buffer Size
- Using TCP.NODELAY
- Using Connection Manager

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

Finding Network 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 network bottlenecks include the following:

- 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 previous example validates trace route. Ideally, you 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 Network Bottlenecks

This section helps you determine the problem with your network bottleneck.

Determining if the Problem is with Oracle Net or the Network

Oracle Net 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 Oracle Net 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_TIMESTAMP_SERVER = ON
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_TIMESTAMP_CLIENT = ON
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_TIMESTAMP_listener_name = ON
TRACE_LEVEL_listener_name = 16
```

Note: The `TRACE_TIMESTAMP_x` parameters are optional, but they should be included for better debugging

---

Reproduce the problem, so that you generate traces on the client and server. Now analyze the traces generated.

See Also:

- *Oracle Net Services Administrator's Guide* for detailed directions on enabling Oracle Net tracing
- *Oracle9i Database Error Messages* for definitions to Oracle Net errors noted in the trace file

If the problem is with the network and not Oracle Net, 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 located, 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 that points to an underlying network problem.

Note: Enabling tracing on the server can generate a large amount of trace files. To prevent this, set up a separate environment that traces itself. This configuration works for dedicated 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, Oracle Net trace analysis takes some time to fully find the problem. However, high-level simple trace analysis is rather simple.

Determining if the Problem is on the Client or the Server (on Oracle Net)

If the problem is with Oracle Net, then use Oracle Net 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.

Checking if the Server is Configured for Shared Servers

The shared server architecture can be more complex to troubleshoot. Check the initialization parameter file for any shared server parameters. Look at the operating system to see if any of the shared server processes are present.

Check for dispatchers by looking for names such as `ora_d000`, `ora_d001`, and so on. For example:

```
ps -ef | grep ora_d
```

Check for shared servers by looking for names such as `ora_s000`, `ora_s001`, and so on. For example:

```
ps -ef | grep ora_s
```

See Also:

- "Shared Server Configuration" on page 23-2 for more information on tuning the shared server
- *Oracle9i Database Concepts* and *Oracle Net Services Administrator's Guide* for more information on shared server concepts and parameters

Using Array Interfaces

Reduce network calls by using array interfaces. Instead of fetching one row at a time, it is more efficient to fetch 10 rows with a single network round trip.

See Also: *Oracle Call Interface Programmer's Guide* for more information on array interfaces

Adjusting Session Data Unit Buffer Size

Before sending data across the network, Oracle Net 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 might 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 Oracle Net Configuration Assistant to configure a change to the default SDU size on both the client and the server; SDU size is generally the same on both.

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

Using TCP.NODELAY

When a session is established, Oracle Net packages and sends data between server and client using packets. The `TCP.NODELAY` parameter, which causes packets to be flushed on to the network more frequently, is enabled by default. Although Oracle Net supports many networking protocols, TCP tends to have the best scalability.

See Also: Your platform-specific Oracle documentation for more information on `TCP.NODELAY`

Using Connection Manager

In Oracle Net, 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. This way, you can increase the number of sessions that a process can handle. This applies only to shared server configurations. Alternately, you can use Connection Manager to control client access to dedicated servers. Connection Manager provides multiple protocol support allowing a client and server with different networking protocols to communicate.

See Also: *Oracle Net Services Administrator's Guide* for more information on Connection Manager

Part VI

Performance-Related Reference Information

Part VI provides reference information regarding dynamic performance views and wait events.

The chapters in this part are:

- Chapter 24, "Dynamic Performance Views for Tuning"

24

Dynamic Performance Views for Tuning

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

Dynamic performance tables are not true tables, and they should not be accessed by most users. However, database administrators can query and create views on the tables and grant access to those views to other users. These views are called fixed views because they cannot be altered or removed by the database administrator.

SYS owns the dynamic performance tables. By default, they are available only to the user SYS and to users granted SELECT ANY TABLE system privilege, such as SYSTEM. Their names all begin with V_$. Views are created on these tables, and then public synonyms are created for the views. The synonym names begin with V$.

This chapter provides detailed information on several of these views that can help you tune your system and investigate performance problems.

Each view belongs to one of the following categories:

- Current State Views
- Counter / Accumulator Views
- Information Views

> **See Also:** *Oracle9i Database Reference* for a complete list of the dynamic performance views and their columns

Current State Views

These views give a picture of what is currently happening on the system.

Table 24–1 Current State Views

Fixed View:	Description:
V$LOCK	Locks currently held / requested on the instance
V$LATCH_HOLDER	Sessions / processes holding a latch
V$OPEN_CURSOR	Cursors opened by sessions on the instance
V$SESSION	Sessions currently connected to the instance
V$SESSION_WAIT	Different resources sessions are currently waiting for

Counter/Accumulator Views

These views keep track of how many times some activity has occurred since instance / session startup. Select from the view directly to see activity since startup.

If you are interested in activity happening in a given time interval, then take a snapshot before and after the time interval, and the delta between the two snapshots provides the activity during that time interval. This is similar to how operating system utilities like sar, vmstat, and iostat work. Tools provided by Oracle, like Statspack and BSTAT / ESTAT, do this delta to provide a report of activity in a given interval.

Note: To avoid the extra overhead incurred as caches are loaded, do not take snapshots immediately after system startup.

Table 24–2 Summary Since Session Startup

Fixed View:	Description:
V$MYSTAT	Resource usage summary for your own session
V$SESSION_EVENT	Session-level summary of all the waits for current sessions
V$SESSTAT	Session-level summary of resource usage since session startup

Table 24–3 Summary Since Instance Startup

Fixed View:	Description:
V$DB_OBJECT_CACHE	Object level statistics in shared pool
V$FILESTAT	File level summary of the I/O activity
V$LATCH	Latch activity summary
V$LATCH_CHILDREN	Aggregate summary for each type of latch
V$LIBRARYCACHE	Namespace level summary for shared pool
V$ROLLSTAT	Rollback activity summary
V$ROWCACHE	Data dictionary activity summary
V$SQL	Child cursor details for V$SQLAREA
V$SQLAREA	Shared pool details for statements/anonymous blocks
V$SYSSTAT	Summary of resource usage
V$SYSTEM_EVENT	Instance wide summary of resources waited for
V$UNDOSTAT	Undo space summary for a ten minute interval
V$WAITSTAT	Break down of buffer waits by class

Information Views

In information views, the information is not as dynamic as in the current state view. Hence, it does not need to be queried as often as the current state views.

Table 24–4 Information Views

Fixed View:	Description:
V$PARAMETER and V$SYSTEM_PARAMETER	Parameters values for your session Instance wide parameter values
V$PROCESS	Server processes (background and foreground)
V$SQL_PLAN	Execution plan for cursors that were recently executed
V$SQLTEXT	SQL text of statements in the shared pool

V$DB_OBJECT_CACHE

This view provides object level statistics for objects in the library cache (shared pool). This view provides more details than V$LIBRARYCACHE and is useful for finding active objects in the shared pool.

Useful Columns for V$DB_OBJECT_CACHE

Most of the columns of this table provide current state information.

- OWNER: Object owner
- NAME: Object name (First 1000 characters of SQL text for anonymous blocks / cursors)
- TYPE: Type of object (for example, sequence, procedure, function, package, package body, trigger)
- KEPT: Tells if the object is pinned in the shared pool (yes, no)
- SHARABLE_MEM: Amount of sharable memory used
- PINS: Sessions currently executing this object
- LOCKS: Sessions currently locking this object

Instantaneous State Columns

The columns below keep statistics on the object since it's first load:

- EXECUTIONS: Number of executions by this object
- LOADS: Number of times this object had to be loaded
- INVALIDATIONS: Number of times this object was invalidated

Example - Summary of Shared Pool Executions and Memory Usage

The following query shows the distribution of the usage (executions) and the shared pool memory across different type of objects. It also shows if any of the objects have been pinned in the shared pool using the procedure DBMS_SHARED_POOL.KEEP().

```
SELECT type, kept, COUNT(*), SUM(executions), SUM(sharable_mem)
  FROM V$DB_OBJECT_CACHE
 GROUP BY kept, type;
```

Example - Finding Objects with Large Number of Loads

```
SELECT owner, name, executions, sharable_mem, kept, loads
FROM V$DB_OBJECT_CACHE
WHERE loads > 1
OR invalidations > 0
ORDER BY loads DESC
```

Example - Finding Large Unpinned Objects

The following query finds all objects using large amounts of memory or executed multiple times. They can be pinned using DBMS_SHARED_POOL.KEEP().

```
SELECT owner, name, executions, sharable_mem, kept
  FROM V$DB_OBJECT_CACHE
 WHERE ( executions > 10
       OR sharable_mem > 102400 )
   AND kept = 'NO'
 ORDER BY 3 desc, 4 DESC
```

V$FILESTAT

This view keeps a summary of physical I/O requests for each file. This is useful in isolating where the I/O activity is happening if the bottleneck is I/O related. V$FILESTAT 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

The numbers reflect activity since the instance startup. If two snapshots are taken, then the differences in the statistics provides the I/O activity for the time interval.

Useful Columns for V$FILESTAT

- FILE#: Number of the file
- PHYRDS: Number of physical reads done
- PHYBLKRD: Number of physical blocks read
- PHYWRTS: Number of physical writes done
- PHYBLKWRT: Number of physical blocks written

Note:

1. Physical reads and blocks read can be different because of multiblock read calls.
2. Physical writes and blocks written can differ because of direct writes by processes.
3. Sum(Physical blocks read) should correlate closely with `physical reads` from V$SYSSTAT.
4. Sum(Physical blocks written) should correlate closely with `physical writes` from V$SYSSTAT.
5. Reads (into buffer cache as well as direct reads) are done by server processes. Writes from buffer cache are handled only by the DBWR. The direct writes are handled by the server processes.

Join Columns for V$FILESTAT

Column:	View:	Joined Column(s):
FILE#	DBA_DATA_FILES	FILE_ID

Example - Checking Oracle Datafile I/O

The following query monitors the values of physical reads and physical writes 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#;
```

The above query also retrieves the name of each datafile from the dynamic performance view V$DATAFILE. Sample output might look like the following:

```
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 Unix file system (UFS) might not reflect true disk accesses. For example, the read times might not 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.

Example - Finding the Files with Large Numbers of Multiblock Reads

The following example is useful for finding tablespaces that might be getting hit by large number of scans.

```
SELECT t.tablespace_name
      ,SUM(a.phyrds-b.phyrds)
       /MAX(86400*(a.snap_date-b.snap_date)) "Rd/sec"
      ,SUM(a.phyblkrd-b.phyblkrd)
       /greatest(SUM(a.phyrds-b.phyrds),1) "Blk/rd"
      ,SUM(a.phywrts-b.phywrts)
       /MAX(86400*(a.snap_date-b.snap_date)) "Wr/sec"
      ,SUM(a.phyblkwrt-b.phyblkwrt)
       /greatest(SUM(a.phywrts-b.phywrts),1) "Blk/wr"
  FROM snap_filestat a, snap_filestat b, dba_data_files t
 WHERE a.file# = b.file#
   AND a.snap_id = b.snap_id + 1
   AND t.file_id = a.file#
 GROUP BY t.tablespace_name
HAVING sum(a.phyblkrd-b.phyblkrd)
       /greatest(SUM(a.phyrds-b.phyrds),1) > 1.1
    OR SUM(a.phyblkwrt-b.phyblkwrt)
       /greatest(SUM(a.phywrts-b.phywrts),1) > 1.1
 ORDER BY 3 DESC, 5 DESC;

TABLESPACE_N Rd/sec Blk/rd Wr/sec Blk/wr
------------ ------ ------ ------ ------
TEMP            2.3   19.7    1.9   24.7
AP_T_02       287.1    7.8     .0    1.0
AP_T_01        12.9    4.0     .2    1.0
APPLSYS_T_01   63.3    2.2     .4    1.0
PO_T_01       313.5    2.1     .2    1.0
RECEIVABLE_T  401.0    1.5    2.4    1.0
SHARED_T_01     9.2    1.3     .4    1.0
SYSTEM         45.2    1.3     .3    1.0
PER_T_01       48.0    1.2     .0     .0
DBA_T_01         .2    1.0     .4    1.4
```

You can see that most of the multiblock reads and writes are going to TEMP tablespace, due to large sorts going to disk. Other tablespaces are getting multiblock reads due to full table scans.

> **See Also:** Chapter 20, "Oracle Tools to Gather Database Statistics" for an example of how to gather file I/O data.

V$LATCH

This view keeps a summary of statistics for each type of latch since instance startup. It is useful for identifying the area within SGA experiencing problems when latch contention is observed in V$SESSION_WAIT.

Useful Columns for V$LATCH

- NAME: Latch name
- IMMEDIATE_GETS: Requests for the latch in immediate mode
- IMMEDIATE_MISSES: IMMEDIATE_GETS that failed
- GETS: Requests for the latch in a willing to wait mode
- MISSES: GETS that did not obtain the latch on first try
- SPIN_GETS: GETS that got the latch within SPIN_GET tries and did not have to sleep
- SLEEP1-SLEEP3: GETS that succeeded only after sleeping one to three times
- SLEEP4: GETS that only succeeded after sleeping four or more times

Join Columns for V$LATCH

Columns:	Fixed View:	Joined Column(s):
NAME	V$LATCH_CHILDREN V$LATCHHOLDER V$LATCHNAME	NAME
NAME	V$LATCH_MISSES	PARENT_NAME
LATCH#	V$LATCH_CHILDREN V$LATCHNAME	LATCH#

V$LATCH Example

In the following example, a table is created to hold data queried from V$LATCH:

```
CREATE TABLE snap_latch as
SELECT 0 snap_id, sysdate snap_date, a.*
  FROM V$LATCH a;
 ALTER TABLE snap_latch add
     ( constraint snap_filestat primary key (snap_id, file#) );
```

```
SELECT name, (a.gets-b.gets)/1000 "Gets(K)",
       a.gets-b.gets)/(86400*(a.snap_date-b.snap_date)) "Get/s",
       100*(a.misses-b.misses)/(a.gets-b.gets) MISS,
       100*(a.spin_gets-b.spin_gets)/(a.misses-b.misses) SPIN,
       (a.gets-b.gets)/1000 "Gets(K)",
       (a.immediate_gets-b.immediate_gets)/1000 "Iget(K)",
       (a.immediate_gets-b.immediate_gets)/
       (86400*(a.snap_date-b.snap_date)) "IGet/s",
       100*(a.immediate_misses-b.immediate_misses)/
       (a.immediate_gets-b.immediate_gets) IMISS
  FROM snap_latch a, snap_latch b
 WHERE a.snap_id = b.snap_id + 1
   AND ( (a.misses-b.misses) > 0.001*(a.gets-b.gets)
       or (a.immediate_misses-b.immediate_misses) >
       0.001*(a.immediate_gets-b.immediate_gets))
ORDER BY 2 DESC;

INSERT INTO snap_latch
SELECT 1, sysdate, a.*
  FROM V$LATCH a;
```

The example shows the latch statistics obtained by doing a delta over a period of one hour (like with the V$FILESTAT numbers). Those latches that had misses less than 0.1% of the gets have bee filtered out.

```
NAME               Gets(K)   Get/s  MISS   SPIN IGets(K)  IGet/s IMISS
------------------ -------- ------- ----- ------ -------- ------- -----
cache buffers chai  255,272  69,938   0.4   99.9    3,902   1,069   0.0
library cache       229,405  62,851   9.1   96.9   51,653  14,151   3.7
shared pool          24,206   6,632  14.1   72.1        0       0   0.0
latch wait list       1,828     501   0.4   99.9    1,836     503   0.5
row cache objects     1,703     467   0.7   98.9    1,509     413   0.2
redo allocation         984     270   0.2   99.7        0       0   0.0
messages                116      32   0.2  100.0        0       0   0.0
cache buffers lru        91      25   0.3   99.0    7,214   1,976   0.3
modify parameter v        2       0   0.1  100.0        0       0   0.0
redo copy                 0       0  92.3   99.3    1,460     400   0.0
```

When examining latch statistics, look at the following:

- What is the ratio of misses / gets?
- What percentage of misses are obtained by just spinning?
- How many times was the latch requested?

There seems to be a lot of contention for the redo copy latch with a 92.3 percent miss rate. But, look carefully. Redo copy latches are obtained mostly in immediate mode. The numbers for immediate gets look fine, and the immediate gets are several orders of magnitude bigger than the willing to wait gets. So, there is no contention for redo copy latches.

However, there does seem to be contention for the shared pool and library cache latches. Consider looking at the sleeps for these latches to see if there is actually a problem.

```
NAME                Gets(K)   Get/s  MISS   SPIN  SL01  SL02  SL03  SL04
------------------ -------- ------- ----- ------ ----- ----- ----- -----
cache buffers chai  255,272  69,938   0.4   99.9   0.1   0.0   0.0   0.0
library cache       229,405  62,851   9.1   96.9   3.0   0.1   0.0   0.0
shared pool          24,206   6,632  14.1   72.1  22.4   4.8   0.8   0.0
latch wait list       1,828     501   0.4   99.9   0.1   0.0   0.0   0.0
row cache objects     1,703     467   0.7   98.9   0.6   0.0   0.4   0.0
redo allocation         984     270   0.2   99.7   0.1   0.0   0.2   0.0
messages                116      32   0.2  100.0   0.0   0.0   0.0   0.0
cache buffers lru        91      25   0.3   99.0   1.0   0.0   0.0   0.0
modify parameter v        2       0   0.1  100.0   0.0   0.0   0.0   0.0
redo copy                 0       0  92.3   99.3   0.0   0.7   0.0   0.0
```

You can see that there is a 14% miss rate on the shared pool latches. 72% of the missed latched without relinquishing the CPU (having to sleep even once) by spinning. There are some misses for which you have to sleep multiple times.

Investigate why the shared pool latch is needed so many times. Look at the SQL being run by sessions holding or waiting for the latch, as well as the resource usage characteristics of the system. Compare them with baselines when there was no problem.

Tuning Latches

Do not tune latches. If you see latch contention, then it is a symptom of a part of SGA experiencing abnormal resource usage. Latches control access with certain assumptions (for example, a cursor is parsed once and executed many times). To fix the problem, examine the resource usage for the parts of SGA experiencing contention. Merely looking at V$LATCH does not address the problem.

> **See Also:** *Oracle9i Database Concepts* for more information on latches

V$LATCH_CHILDREN

There are multiple latches in the database for some type of latches. V$LATCH provides aggregate summary for each type of latch. To look at individual latches, query V$LATCH_CHILDREN.

Example - Finding the Number of Multiple Latches on the System

```
SELECT name, COUNT(*) FROM V$LATCH_CHILDREN
 GROUP BY name ORDER BY 2 DESC;

NAME                                     COUNT(*)
---------------------------------------- ----------
global tx hash mapping                         2888
global transaction                             2887
cache buffers chains                           2048
latch wait list                                  32
Token Manager                                    23
enqueue hash chains                              22
session idle bit                                 22
redo copy                                        22
process queue reference                          20
Checkpoint queue latch                           11
library cache                                    11
msg queue latch                                  11
session queue latch                              11
process queue                                    11
cache buffers lru chain                          11
done queue latch                                 11
channel operations parent latch                   4
session switching                                 4
message pool operations parent latch              4
ksfv messages                                     2
parallel query stats                              2
channel handle pool latch                         1
temp table ageout allocation latch                1
```

V$LATCH_HOLDER

This view is useful to see if the session holding the latch is changing. Most of the time, the latch is held for such a small time that it is impossible to join to some other table to see the SQL statement being executed or the events that latch holder is waiting for.

The main use for this view is to see that the latch is not stuck on some session.

Join Columns for V$LATCH_HOLDER

Columns:	Fixed View:	Joined Column(s):
LADDR	V$LATCH_CHILDREN	ADDR
NAME	V$LATCH, V$LATCHNAME, V$LATCH_CHILDREN	NAME
PID	V$PROCESS	PID
SID	V$SESSION	SID

Example - Finding the SQL Statement Executed by the Latch Holder

```
SELECT s.sql_hash_value, l.name
  FROM V$SESSION s, V$LATCHHOLDER l
WHERE s.sid = l.sid;

SQL_HASH_VALUE NAME
-------------- ------------------------------------------
     299369270 library cache
    1052917712 library cache
    3198762001 library cache
SQL> /

SQL_HASH_VALUE NAME
-------------- ------------------------------------------
     749899113 cache buffers chains
    1052917712 library cache
SQL> /

SQL_HASH_VALUE NAME
-------------- ------------------------------------------
    1052917712 library cache
SQL> /
```

```
SQL_HASH_VALUE NAME
-------------- ------------------------------------------
     749899113 library cache
    1052917712 library cache
```

This example indicates that the SQL statement 1052917712 is using a lot of parsing resources. The next step is to find the resources used by the session and examine the statement.

V$LIBRARYCACHE

This view has a namespace level summary for the objects in library cache since instance startup. When experiencing performance issues related to the library cache, this view can help identify the following:

- Specific parts (namespace) of the library cache (shared pool)
- Possible causes of problems

Then use VDB_OBJECT_CACHE, VSQLAREA to get more details.

Useful Columns for V$LIBRARYCACHE

- NAMESPACE: Class of objects (SQL area, trigger, and so on)
- GETS: Handle requests for objects of this namespace
- GETHITS: Requests that found handle in the cache
- PINS: PIN requests for objects of this namespace
- PINHITS: Requests able to reuse an existing PIN
- RELOADS: Number of times objects stored in the library cache had to be reloaded into memory because part of the object had been flushed from the cache. If there are a significant number of reloads, then reusable information is being flushed from the library cache. This requires a reload / rebuild of the object before it can again be accessed.
- INVALIDATIONS: The number of times objects were invalidated. For example, an object is invalidated automatically by Oracle when it is no longer safe to execute. If the optimizer statistics for a table were recomputed, then all SQL statements currently in the library cache at the time the recompute occurred would be invalidated, because their execution plans may no longer be optimal.

GETHITRATIO (GETHITS / GETS) and GETPINRATIO (PINHITS / PINS) can be used if just examining activity since instance startup. If examining activity over a specified time interval, it is better to compute these from the differences in snapshots before and after the interval.

Example - V$LIBRARYCACHE Query

```
SELECT namespace, gets, 100*gethits/gets gethitratio,
       pins, 100* pinhits/pins getpinratio,
       reloads, invalidations
  FROM V$LIBRARYCACHE
 ORDER BY gets DESC
```

Look for the following when querying this view:

- High RELOADS or INVALIDATIONS
- Low GETHITRATIO or GETPINRATIO

High number of RELOADS could be due to the following:

- Objects being invalidated (large number of INVALIDATIONS)
- Objects getting swapped out of memory

Low GETHITRATIO could indicate that objects are getting swapped out of memory.

Low PINHITRATIO could indicate the following:

- Session not executing the same cursor multiple times (even though it might be shared across different sessions)
- Session not finding the cursor shared

The next step is to query VDB_OBJECT_CACHE / VSQLAREA to see if problems are limited to certain objects or spread across different objects. If invalidations are high, then it might be worth investigating which of the (invalidated object's) underlying objects are being changed.

V$LOCK

This view has a row for every lock held or requested on the system. You should examine this view if you find sessions waiting for the wait event *enqueue*. If you find sessions waiting for a lock, then the sequence of events could be the following:

1. Use V$LOCK to find the sessions holding the lock.
2. Use V$SESSION to find the SQL statements being executed by the sessions holding the lock and waiting for the lock.
3. Use V$SESSION_WAIT to find what the session holding the lock is doing.
4. Use V$SESSION to get more details about the program and user holding the lock.

Useful Columns for V$LOCK

- SID: Identifier of the session holding / requesting the lock
- TYPE: Type of lock
- LMODE: The mode the lock is held in
- REQUEST: The mode the lock is requested in
- ID1, ID2: Lock resource identifiers

Common Lock Types

TX: Row Transaction Lock

- This lock is required in exclusive mode to change any data.
- One lock is acquired for each active transaction. It is released when the transaction ends due to a commit or rollback.
- If a block containing the row(s) to be changed does not have any ITL (interested transaction list) entries left, then the session requests the lock in shared mode. It is released when the session gets an ITL entry for the block.
- If any of the rows to be changed are locked by another session, then locking session's transaction lock is requested in exclusive mode. When the locking transaction ends, this request ends, and the rows are covered under the requesting session's existing TX lock.
- The lock points to the rollback segment and transaction table entries for the transaction.

TM: DML Lock

- This lock is required in exclusive mode for executing any DDL statements on a database object; for example, lock table in exclusive mode, alter table, drop table.
- This lock is also acquired in shared mode by any statement changing any data to prevent any DDL on the object.
- For every object whose data is being changed, a TM lock is required.
- The lock points to the object.

ST - Space Transaction Lock

- There is only one lock per database (not instance).
- This lock is required in exclusive mode for any space management activity (creation or dropping any extents) except with locally managed tablespaces.
- Object creation, dropping, extension, and truncation all serialize on this lock.
- Most common causes for contention on this lock are sorting to disk (not using true temporary tablespaces) or rollback segment extension and shrinking.

Do the following to avoid contention on this enqueue:

- Use true temporary tablespaces (temporary segments are not created and dropped after every sort to disk).
- Use locally managed tablespaces with uniform extents.
- Size rollback segments to avoid dynamic extension and shrinking.
- Avoid application practices that create and drop database objects.

UL - User Defined Locks

Users can define their own locks.

> **See Also:** *Oracle9i Database Concepts* for more information on locks

Common Modes for Request/Lmode

- 0: None
- 2: Row Share: used for shared DML locks
- 4: Share: used for shared TX when waiting for ITL entry

- 6: Exclusive used for row level, DML locks

Any row in V$LOCK either has LMODE=0 (indicating it is a request) or REQUEST=0 (indicating it is a held lock).

Resource Identifier ID1

For DML locks, ID1 is the object_id.

For TX locks, ID1 points to the rollback segment and transaction table entry.

Join Columns for V$LOCK

Columns:	Fixed View:	Joined Column(s):
SID	V$SESSION	SID
ID1, ID2, TYPE	V$LOCK	ID1, ID2, TYPE
ID1	DBA_OBJECTS	OBJECT_ID
TRUNC(ID1/65536)	V$ROLLNAME	USN

1. This is used to find the session holding the lock, if a session is waiting for a lock.
2. This can be used to find the locked object for DML locks (type = 'TM').
3. This can be used to find the rollback segment in use for row transaction locks (TYPE = 'TX'). However, a less cryptic join might be via V$TRANSACTION.

Example - Finding the Sessions Holding the Lock

Find the (ID1, ID2, type) for sessions waiting for a lock (LMODE=0).

Find the session holding the lock (REQUEST=0) for that ID1, ID2, type.

```
SELECT lpad(' ',DECODE(request,0,0,1))||sid sess, id1,id2,lmode
      ,request, type
FROM V$LOCK
 WHERE id1 IN (SELECT id1 FROM V$LOCK WHERE lmode = 0)
 ORDER BY id1,request

SID          ID1        ID2      LMODE    REQUEST TY
------ ---------- ---------- ---------- ---------- --
1237       196705     200493          6          0 TX <- Lock Holder
 1256      196705     200493          0          6 TX <- Lock Waiter
 1176      196705     200493          0          6 TX <- Lock Waiter
938        589854     201352          6          0 TX <- Lock Holder
```

```
 1634      589854     201352          0          6 TX <- Lock Waiter
```

Example - Finding the Statements being Executed by These Sessions

```
SELECT sid, sql_hash_value
FROM V$SESSION
WHERE SID IN (1237,1256,1176,938,1634);

SID  SQL_HASH_VALUE
-----  --------------
  938      2078523611 <-Holder
 1176      1646972797 <-Waiter
 1237      3735785744 <-Holder
 1256      1141994875 <-Waiter
 1634      2417993520 <-Waiter
```

Example - Finding the Text for These SQL Statements

```
HASH_VALUE SQL_TEXT
---------- ------------------------------------------------------------------
1141994875 SELECT TO_CHAR(CURRENT_MAX_UNIQUE_IDENTIFIER + 1 )   FROM PO_UNI
           QUE_IDENTIFIER_CONTROL  WHERE TABLE_NAME = DECODE(:b1,'RFQ','PO_
           HEADERS_RFQ','QUOTATION','PO_HEADERS_QUOTE','PO_HEADERS') FOR UP
           DATE OF CURRENT_MAX_UNIQUE_IDENTIFIER
1646972797 SELECT TO_CHAR(CURRENT_MAX_UNIQUE_IDENTIFIER + 1 )   FROM PO_UNI
           QUE_IDENTIFIER_CONTROL  WHERE TABLE_NAME = 'PO_HEADERS'  FOR UPD
           ATE OF CURRENT_MAX_UNIQUE_IDENTIFIER
2078523611 select CODE_COMBINATION_ID,  enabled_flag,  nvl(to_char(start_da
           te_active, 'J'), -1),  nvl(to_char(end_date_active, 'J'), -1), S
           EGMENT2||'.'||SEGMENT1||'.'||||SEGMENT6,detail_posting_allowed_f
           lag,summary_flag  from GL_CODE_COMBINATIONS  where CHART_OF_ACCO
           UNTS_ID = 101  and SEGMENT2 in ('000','341','367','388','389','4
           52','476','593','729','N38','N40','Q21','Q31','U21')  order by S
           EGMENT2, SEGMENT1, SEGMENT6
2417993520 select 0 into :b0  from pa_projects where project_id=:b1 for upd
           ate
3735785744 begin :X0 := FND_ATTACHMENT_UTIL_PKG.GET_ATCHMT_EXISTS(:L_ENTITY
           _NAME, :L_PKEY1, :L_PKEY2, :L_PKEY3, :L_PKEY4, :L_PKEY5, :L_FUNC
           TION_NAME, :L_FUNCTION_TYPE); end;
```

The locked sessions' statements show that the sessions 1176 and 1256 are waiting for a lock on the PO_UNIQUE_IDENTIFIER_CONTROL held by session 1237, while session 1634 is waiting for a lock on PA_PROJECTS held by session 938. Query V$SESSION_WAIT, V$SESSION, and V$SESSION_EVENT to get more details about the sessions and users. For example:

- Who is holding the lock?
- Is the session holding the lock active, or has the user gone to lunch?
- Is the session executing long running queries while holding the lock?

V$MYSTAT

This view is a subset of V$SESSTAT returning current session's statistics. When auditing resource usage for sessions via triggers, use V$MYSTAT to capture the resource usage, because it is much cheaper than scanning the rows in V$SESSTAT.

V$OPEN_CURSOR

This view lists all the cursors opened by the sessions. There are several ways it can be used. For example, you can monitor the number of cursors opened by different sessions.

When diagnosing system resource usage, it is useful to query V$SQLAREA and V$SQL for expensive SQL (high logical or physical I/O). In such cases, the next step is to find it's source. On applications where users log in to the database as the same generic user (and have the same PARSING_USER_ID in V$SQLAREA), this can get difficult. The statistics in V$SQLAREA are updated after the statement completes execution (and disappears from V$SESSION.SQL_HASH_VALUE). Therefore, unless the statement is being executed again, you cannot find the session directly. However, if the cursor is still open for the session, then use V$OPEN_CURSOR to find the session(s) that have executed the statement.

Join Columns for V$OPEN_CURSOR

Columns:	Fixed View:	Joined Column(s):
HASH_VALUE, ADDRESS	V$SQLAREA, V$SQL, V$SQLTEXT	HASH_VALUE, ADDRESS
SID	V$SESSION	SID

Example 2 - Find the Session(s) that Executed a Statement

```
SELECT hash_value, buffer_gets, disk_reads
FROM V$SQLAREA
WHERE disk_reads > 1000000
ORDER BY buffer_gets DESC;

HASH_VALUE BUFFER_GETS DISK_READS
---------- ----------- ----------
1514306888   177649108    3897402
 478652562    63168944    2532721
 360282550    14158750    2482065
 226079402    40458060    1592621
2144648214     1493584    1478953
1655760468     1997868    1316010
 160130138     6609577    1212163
3000880481     2122483    1158608

8 rows selected.
```

```
SQL> SELECT sid FROM V$SESSION WHERE sql_hash_value = 1514306888 ;

no rows selected

SQL> SELECT sid FROM V$OPEN_CURSOR WHERE hash_Value = 1514306888 ;

  SID
-----
 1125
  233
  935
 1693
  531

5 rows selected.
```

Example 1 - Finding Sessions that have more than 400 Cursors Open

```
SELECT sid, COUNT(*)
FROM V$OPEN_CURSOR
 GROUP BY sid
HAVING COUNT(*) > 400
 ORDER BY 2 DESC;

 SID   COUNT(*)
----- ----------
 2359        456
 1796        449
 1533        445
 1135        442
 1215        442
  810        437
 1232        429
   27        426
 1954        421
 2067        421
 1037        416
 1584        413
  416        407
  398        406
  307        405
 1545        403
```

V$PARAMETER and V$SYSTEM_PARAMETER

These views list each initialization parameter by name and show the value for that parameter. The V$PARAMETER view shows the current value for the session performing the query. The V$SYSTEM_PARAMETER view shows the instance-wide value for the parameter.

For example, executing the following query shows the SORT_AREA_SIZE parameter setting for the session executing the query:

```
SELECT value
  FROM V$PARAMETER
 WHERE name = 'sort_area_size';
```

Useful Columns for V$PARAMETER

- NAME: Name of the parameter
- VALUE: Current value for this session (if modified within the session); otherwise, the instance-wide value
- ISDEFAULT: Whether this parameter has been specified by the user as an initialization parameter
- ISSES_MODIFIABLE: Whether this parameter can be modified at the session level
- ISSYS_MODIFIABLE: Whether this parameter can be modified at an instance-wide level dynamically after the instance has started
- ISMODIFIED: Whether this parameter has been modified after instance startup, and if so, whether it was modified at the session level or at the instance (system) level
- ISADJUSTED: Whether Oracle has adjusted a value specified by the user
- DESCRIPTION: Brief description of the parameter
- UPDATE_COMMENT: Set if a comment has been supplied by the DBA for this parameter

See Also:

- *Oracle9i Database Reference* for more information on the range column values
- *Oracle9i Database Administrator's Guide* for information on server parameter files

Uses for V$PARAMETER and V$SYSTEM_PARAMETER Data

V$PARAMETER is queried during performance tuning to determine the current settings for a parameter. For example, if the buffer cache hit ratio is low, then the value for DB_BLOCK_BUFFERS (or DB_CACHE_SIZE) can be queried to determine the current buffer cache size.

The SHOW PARAMETER statement in SQL*Plus queries data from V$PARAMETER.

Example - Determining the SORT_AREA_SIZE from within SQL*Plus

```
column name             format a20
column value            format a10
column isdefault        format a5
column isses_modifiable format a5

SELECT name, value, isdefault, isses_modifiable, issys_modifiable, ismodified
  FROM V$PARAMETER
 WHERE name = 'sort_area_size';

NAME                 VALUE      ISDEF ISSES ISSYS_MOD ISMODIFIED
-------------------- ---------- ----- ----- --------- ----------
sort_area_size       1048576    TRUE  TRUE  DEFERRED  MODIFIED
```

The above example shows that the SORT_AREA_SIZE initialization parameter was not set as an initialization parameter on instance startup, but was modified at the session level (indicated by the ISMODIFIED column having the value of MODIFIED) for this session.

Note: Use caution when querying from V$PARAMETER. If you want to see the instance-wide parameters, use V$SYSTEM_PARAMETER view instead of V$PARAMETER.

V$PROCESS

This view contains information about all Oracle processes running on the system. It is used to relate the Oracle or operating system process ID of the server process to the database session. This is needed in several situations:

- If the bottleneck on the database server is related to an operating system resource (for example, CPU, memory), then in most of the cases the processes using most of the resources are the server processes. In such a scenario, the next steps are as follows:
  1. Find the resource intensive processes.
  2. Find their sessions. You must relate the processes to sessions.
  3. Find out why the session is using so many resources.
- The SQL*Trace file names are based on the operating system process ID of the server process. To locate the trace file for a session, you must relate the session to the server process.
- Some events, like `rdbms ipc reply`, identify the Oracle process ID of the process a session is waiting on. To find out what those processes are doing, you must find their sessions.
- The background processes you see on the server (DBWR, LGWR, PMON, and so on) are all server processes. To see what they are doing in the database, you must find their session.

Useful Columns for V$PROCESS

- `PID`: Oracle process ID of the process
- `SPID`: Operating system process ID of the process

Join Columns for V$PROCESS

Column:	Fixed View:	Joined Column(s):
ADDR	V$SESSION	PADDR

Example 1 - Finding the Session for Server Process 20143

```
SELECT ' sid, serial#, aud sid : '|| s.sid||' , '||s.serial#||' , '||
       s.audsid||chr(10)|| '     DB User / OS User : '||s.username||
       '   /   '||s.osuser||chr(10)|| '    Machine - Terminal : '||
       s.machine||'  -  '|| s.terminal||chr(10)||
       '        OS Process Ids : '|| s.process||' (Client)  '||
       p.spid||' - '||p.pid||' (Server)'|| chr(10)||
       '   Client Program Name : '||s.program "Session Info"
  FROM V$PROCESS P,V$SESSION s
 WHERE p.addr = s.paddr
   AND p.spid = '20143';

Session Info
--------------------------------------------------------------------
 Sid, Serial#, Aud sid : 2204 , 5552 , 14478782
     DB User / OS User : APPS   /   sifapmgr
    Machine - Terminal : finprod3  -
        OS Process Ids : 9095 (Client)  20143 - 1404 (Server)
   Client Program Name : RGRARG@finprod3 (TNS V1-V3)
```

Example 2 - Find the Session for PMON

```
SELECT ' sid, serial#, aud sid : '|| s.sid||' , '||s.serial#||' , '||
       s.audsid||chr(10)|| '     DB User / OS User : '||s.username||
       '   /   '||s.osuser||chr(10)|| '    Machine - Terminal : '||
       s.machine||'  -  '|| s.terminal||chr(10)||
       '        OS Process Ids : '|| s.process||' (Client)  '||
       p.spid||' - '||p.pid||' (Server)'|| chr(10)||
       '   Client Program Name : '||s.program "Session Info"
  FROM V$PROCESS p, V$SESSION s
 WHERE p.addr = s.paddr
   AND s.program LIKE '%PMON%'

Session Info
------------------------------------------------------------
 Sid, Serial#, Aud sid : 1 , 1 , 0
     DB User / OS User :    /   oracle
    Machine - Terminal : finprod7  -  UNKNOWN
        OS Process Ids : 20178 (Client)  20178 - 2 (Server)
   Client Program Name : oracle@finprod7 (PMON)
```

You can see that the client and server processes are the same for the background process, which is why we could specify the client program name.

V$ROLLSTAT

This view keeps a summary of statistics for each rollback segment since startup.

Useful Columns for V$ROLLSTAT

- USN: Rollback segment number
- RSSIZE: Current size of the rollback segment
- XACTS: Number of active transactions

Columns Useful for doing a Delta over a Period of Time

- WRITES: Number of bytes written to the rollback segment
- SHRINKS: Number of times the rollback segment grew past OPTIMAL and shrank back
- EXTENDS: Number of times the rollback segment had to extend because there was an active transaction in the next extent
- WRAPS: Number of times the rollback segment wrapped around
- GETS: Number of header gets
- WAITS: Number of header waits

Join Columns for V$ROLLSTAT

Column:	Fixed View:	Joined Column(s):
USN	V$ROLLNAME	USN

Example - V$ROLLSTAT

By dividing the elapsed time by wraps, you can determine the average time taken for a rollback segment to wrap. This is useful in sizing rollback segments for long running queries to avoid 'Snapshot Too Old' errors.

Also, monitor the extends and shrinks to see if the optimal size should be increased.

V$ROWCACHE

This view displays statistics for the dictionary cache (also known as the rowcache). Each row contains statistics for the various types of dictionary cache data. Note that there is a hierarchy in the dictionary cache, so the same cache name can appear more than once.

Useful Columns for V$ROWCACHE

- PARAMETER: Name of the cache
- COUNT: Number of entries allocated to this cache
- USAGE: Current number of used entries
- GETS: Total number of requests
- GETMISSES: Number of requests resulting in dictionary cache miss
- SCANS: Number of scan requests
- SCANMISSES: Number of times a scan failed to find the required data
- MODIFICATIONS: Number of additions, changes or deletions of cache entries
- DLM_REQUESTS: Number of DLM Real Application Clusters requests
- DLM_CONFLICTS: Number of DLM Real Application Clusters conflicts
- DLM_RELEASES: Number of DLM Real Application Clusters releases

Uses for V$ROWCACHE Data

- Determine whether the dictionary cache is adequately sized. If the shared pool is too small, then the dictionary cache is not able to grow to a sufficient size to cache the required information.

  See Also: Chapter 24, "Dynamic Performance Views for Tuning" for details on tuning the shared pool

- Determine whether the application is accessing the cache efficiently. If the application design uses the dictionary cache inefficiently (in this case, a larger dictionary cache will not alleviate the performance problem). For example, if a large number of GETS appear for the DC_USERS cache within the sample period, then it is likely that there are large number of distinct users created within the database, and that the application is logging the users on and off frequently. To verify this, check the logon rate and also the number of users in the system. The parse rates will also be high. If this is a large OLTP system with

a middle tier, then it might be more efficient to manage individual accounts on the middle tier, allowing the middle tier to logon as a single use: the application owner. Reducing logon/logoff rate by keeping connections alive also helps.

- Determine whether dynamic space allocation is occurring. A large number of similarly sized modifications for DC_SEGMENTS, DC_USED_EXTENTS, and DC_FREE_EXTENTS can indicate much dynamic space allocation, in which case the solution is to size next extents appropriately.
- Identify large amounts of sequence number generation occurring. Modifications to dc_sequences indicates this. Check to see whether the number of cache entries per sequence number are sufficient for then number of changes.
- Gather evidence for hard parsing. Hard parsing can also be evidenced by many GETS to DC_COLUMNS, DC_VIEWS and DC_OBJECTS caches.

Example - Viewing V$ROWCACHE Data

A good way to view dictionary cache statistics is to group the data by the cache name.

```
SELECT parameter
     , sum("COUNT")
     , sum(usage)
     , sum(gets)
     , sum(getmisses)
     , sum(scans)
     , sum(scanmisses)
     , sum(modifications)
     , sum(dlm_requests)
     , sum(dlm_conflicts)
     , sum(dlm_releases)
  FROM V$ROWCACHE
 GROUP BY parameter;
```

V$SESSION

This view has one row for every session connected to the database instance. The sessions include user sessions, as well as background processes like DBWR, LGWR, archiver.

See Also: *Oracle9i Database Concepts*

Useful Columns for V$SESSION

V$SESSION is basically an information view used for finding the SID or SADDR of a user. However, it has some columns that change dynamically and are useful for examining a user. For example:

SQL_HASH_VALUE, SQL_ADDRESS: These identify the SQL statement currently being executed by the session. If NULL or 0, then the session is not executing any SQL statement. PREV_HASH_VALUE and PREV_ADDRESS identify the previous statement being executed by the session.

Note: When selecting from SQL*Plus, make sure that you have the column defined with adequate width (11 numbers wide) to see the complete number.

STATUS: This column identifies if the session is:

- Active: executing a SQL statement (waiting for / using a resource)
- Inactive: waiting for more work (that is, SQL statements)
- Killed: marked to be killed

The following columns provide information about the session and can be used to find a session when a combination (one or more) of the following are known:

Session Information

- SID: Session identifier, used to join to other columns
- SERIAL#: Counter, which is incremented each time a SID is reused by another session (when a session ends and another session starts and uses the same SID)
- AUDSID: Auditing session ID uniquely identifies a session over the life of a database. It is also useful when finding the parallel query slaves for a query coordinator (during the PQ execution they have the same AUDSID)
- USERNAME: The Oracle user name for the connected session

Client Information

The database session is initiated by a client process that could be running on the database server or connecting to the database across SQL*Net from a middle tier server or even a desktop. The following columns provide information about this client process:

- OSUSER: Operating system user name for the client process
- MACHINE: Machine where the client process is executing
- TERMINAL: Terminal (if applicable) where the client process is running
- PROCESS: Process ID of the client process
- PROGRAM: Client program being executed by the client process

To display TERMINAL, OSUSER for users connecting from PCs, set the keys TERMINAL, USERNAME in ORACLE.INI or the Windows registry on their PCs if they are not showing up by default.

Application Information

Call the package DBMS_APPLICATION_INFO to set some information to identify the user. This shows up in the following columns:

- CLIENT_INFO: Set in DBMS_APPLICATION_INFO
- ACTION: Set in DBMS_APPLICATION_INFO
- MODULE: Set in DBMS_APPLICATION_INFO

Join Columns for V$SESSION

This is a list of columns that we can use to join to other fixed views.

Columns:	Fixed View:	Joined Column(s):
SID	V$SESSION_WAIT, V$SESSTAT, V$LOCK, V$SESSION_EVENT, V$OPEN_CURSOR	SID
(SQL_HASH_VALUE, SQL_ADDRESS)	V$SQLTEXT, V$SQLAREA, V$SQL	(HASH_VALUE, ADDRESS)
(PREV_HASH_VALUE, PREV_SQL_ADDRESS)	V$SQLTEXT, V$SQLAREA, V$SQL	(HASH_VALUE, ADDRESS)
TADDR	V$TRANSACTION	ADDR
PADDR	V$PROCESS	ADDR

Example 1 - Finding your Session

```
SELECT SID, OSUSER, USERNAME, MACHINE, PROCESS
  FROM V$SESSION
 WHERE audsid = userenv('SESSIONID');

  SID OSUSER     USERNAME    MACHINE     PROCESS
----- ---------- ----------- ----------- --------
 1011 vsaksena   SYSTEM      dlsun1653   15912
```

Example 2 - Finding a Session When the Machine is Known

```
SELECT SID, OSUSER, USERNAME, MACHINE, TERMINAL
  FROM V$SESSION
 WHERE terminal = 'ttyAH/AHHh'
   AND machine  = 'prodseq4';

 SID OSUSER    USERNAME  MACHINE    TERMINAL
---- --------- --------- ---------- ----------
  58 vsaksena  APPS_US   prodseq4   ttyAH/AHHh
```

Example 3 - SQL Statement Currently Being Run by a Session

It is a common requirement to find the SQL statement currently being executed by a given session. If a session is experiencing or responsible for a bottleneck, then the statement explains what the session might be doing.

```
col hash_value form 99999999999
SELECT sql_hash_value hash_value
  FROM V$SESSION WHERE sid = 406;

HASH_VALUE
----------
4249174653
SQL> /

HASH_VALUE
----------
4249174653
SQL> /

HASH_VALUE
----------
4249174653
SQL> /
```

```
HASH_VALUE
----------
4249174653
```

This example waited for five seconds, executed the statement again, and repeated the action couple of times. The same hash_value comes up again and again, indicating that the statement is being executed by the session. As a next step, find the statement text using the view V$SQLTEXT and statement statistics from V$SQLAREA.

V$SESSION_EVENT

This view summarizes wait events for every session. While V$SESSION_WAIT shows the current waits for a session, V$SESSION_EVENT provides summary of all the events the session has waited for since it started.

Useful Columns for V$SESSION_EVENT

- SID: Identifier for the session
- EVENT: Name of the wait event
- TOTAL_WAITS: Total number of waits for this event by this session
- TIME_WAITED: Total time waited for this event (in hundredths of a second)
- AVERAGE_WAIT: Average amount of time waited for this event by this session (in hundredths of a second)
- TOTAL_TIMEOUTS: Number of times the wait timed out

Join Columns for V$SESSION_EVENT

Column:	Fixed View:	Joined Column(s):
SID	V$SESSION	SID

Example - Finding the Waits for the Database Writer

```
SELECT s.sid, bgp.name
    FROM V$SESSION s, V$BGPROCESS bgp
   WHERE bgp.name LIKE '%DBW%'
     AND bgp.paddr = s.paddr;

SELECT event, total_waits waits, total_timeouts timeouts,
       time_waited total_time, average_wait avg
  FROM V$SESSION_EVENT
 WHERE sid = 3
 ORDER BY 4 DESC;
```

```
EVENT                          WAITS TIMEOUTS TOTAL_TIME   AVG
------------------------------ -------- -------- ---------- -----
rdbms ipc message             1684385   921495  284706709 169.03
db file parallel write         727326        0    3012982   4.14
latch free                        157      157        281   1.78
control file sequential read      123        0         61   0.49
file identify                      45        0         29   0.64
direct path read                   41        0          5   0.12
file open                          49        0          2   0.04
db file sequential read             2        0          2   1.00
```

V$SESSION_WAIT

This is a key view for finding bottlenecks. It tells what every session in the database is currently waiting for (or the last event waited for by the session if it is not waiting for anything). This view can be used as a starting point to find which direction to proceed in when a system is experiencing performance problems.

V$SESSION_WAIT has a row in V$SESSION_WAIT for every session connected to an instance. It indicates if the session is:

- Using a resource
- Waiting for a resource
- Idle (waiting on one of the idle events)

Useful Columns for V$SESSION_WAIT

- SID: Session identifier for the session
- EVENT: Event the session is currently waiting for, or the last event the session had to wait for
- WAIT_TIME: Time (in hundredths of a second) that the session waited for the event; if the WAIT_TIME is 0, then the session is currently waiting for the event
- SEQ#: Gets incremented with every wait of the session
- P1, P2, P3: Wait event specific details for the wait
- P1TEXT, P2TEXT, P3TEXT: Description of P1,P2,P3 for the given event

> **See Also:** *Oracle9i Database Reference* and "Wait Events" on page 22-19

Table 24–5 Wait Time Description

WAIT-TIME:	Meaning:	Waiting
>0	Time waited in the last wait (in 10 ms clock ticks)	No
0	Session is currently waiting for this event	Yes
-1	Time waited in the last wait was less than 10 ms	No
-2	Timing is not enabled	No

Below is an example of how the EVENT, SEQ#, and WAIT_TIME might change over a period of time:

Table 24–6 Events Changing Over Time

Time	Seq #	Event	Wait Time	P1	P2	P3	Action	Waiting
0	43	latch free	0	800043F8	31	1	Get LRU latch	Yes
10	43	latch free	10	800043F8	31	1	Get free buffer	No
20	44	db file sequential read	0	5	1345	1	Issue the read call	Yes
30	44	db file sequential read	10	5	1345	1	Process the buffer	No
35	45	enqueue	0	1415053318	196631	6355	Lock the buffer	Yes
1040	45	enqueue	1000	1415053318	196631	6355	Modify the buffer	No

In this example, the session waited for a *latch* from 0-10, waited for *db file sequential read* from 20-30, waited for a *lock* from 35-1040. The times in between have been exaggerated for illustration purposes. Event and Seq# do not change until the session has to wait again. The Wait Time indicates if the session is actually waiting or using a resource.

Join Columns for V$SESSION_WAIT

Column:	Fixed View:	Joined Column(s):
SID	V$SESSION	SID

Example - Finding Current Waits on the System

```
SELECT event, SUM(DECODE(wait_time,0,1,0)) "Curr", SUM(DECODE(wait_time,0,0,1))
"Prev", COUNT(*)"Total"
FROM V$SESSION_WAIT
GROUP BY event ORDER BY 4;
```

```
EVENT                                      Prev  Curr   Tot
------------------------------------------ ---- ----- -----
PL/SQL lock timer                             0     1     1
SQL*Net more data from client                 0     1     1
smon timer                                    0     1     1
pmon timer                                    0     1     1
SQL*Net message to client                     2     0     2
db file scattered read                        2     0     2
rdbms ipc message                             0     7     7
enqueue                                       0    12    12
pipe get                                      0    12    12
db file sequential read                       3    10    13
latch free                                    9     6    15
SQL*Net message from client                 835  1380  2215
```

This query, which groups the data by event and by wait_time (0=waiting, nonzero=not waiting), shows the following:

- Most of the sessions are waiting for idle events like `SQL*Net message from client`, `pipe get`, `PMON timer`, and so on.
- The number of sessions using the CPU can be approximated by the number of sessions not waiting (prev), except for one problem: there seem to be a lot of sessions that are not waiting for anything (hence actively using resources) and whose last wait was `SQL*Net message from client`.

The next step should be to check `V$SESSION` to see if the session is active or not. Only count the session as actively waiting or using a resource if it is active. Use the statement below to accomplish this. The total column counts the total of all the sessions, however the currently waiting and previously waited (using resource) columns only count active sessions.

```
SELECT event,
       SUM(DECODE(wait_Time,0,0,DECODE(s.status,'ACTIVE',1,0))) "Prev",
       SUM(DECODE(wait_Time,0,1,DECODE(s.status,'ACTIVE',1,0))) "Curr",
       COUNT(*) "Tot"
  FROM V$SESSION S,V$SESSION_WAIT  w
 WHERE s.sid = w.sid
 GROUP BY event ORDER BY 4;
```

```
EVENT                                    Prev  Curr   Tot
---------------------------------------- ----- ----- -----
SQL*Net message to client                   1     1     1 <- idle event
buffer busy waits                           1     1     1
file open                                   1     1     1
pmon timer                                  0     1     1 <- idle event
smon timer                                  0     1     1 <- idle event
log file sync                               0     1     1
db file scattered read                      0     2     2
rdbms ipc message                           0     7     7 <- idle event
pipe get                                    0    12    12 <- idle event
enqueue                                     0    14    14
latch free                                 10    17    20
db file sequential read                     7    22    23
SQL*Net message from client                 0  1383  2240 <- idle event
```

Now sessions are counted as actively waiting or using a resource only if they are active. This highlights the following:

- There are a total of 2324 sessions.
- 20 sessions are actively using resources (active sessions without an active wait).
- 1463 sessions are waiting.
- 58 of these are waiting for nonidle events. The idle events here being `SQL*Net message from client`, `pipe get`, `rdbms ipc message`, `PMON timer`, `SMON timer`, and `SQL*Net message to client`.

  See Also: "Wait Events" on page 22-19

- 14 sessions are locked out (and probably screaming about poor performance).
- PMON and SMON are sleeping on their timers.
- 24 sessions are waiting for I/O calls to return (db file%read).

V$SESSTAT

V$SESSTAT stores *session*-specific resource usage statistics, beginning at login and ending at logout.

Similar to V$SYSSTAT, this view stores the following types of statistics:

- A count of the number of times an action occurred (`user commits`)
- A running total of volumes of data generated, accessed or manipulated (`redo size`)
- If `TIMED_STATISTICS` is true, the cumulative time spent performing some actions (`CPU used by this session`)

The differences between V$SYSSTAT and V$SESSTAT are the following:

- V$SESSTAT only stores data on a per session basis, whereas V$SYSSTAT stores the accumulated values for all sessions.
- V$SESSTAT is transitory, and is lost after a session logs out. V$SYSSTAT is cumulative, and is only lost when the instance is shutdown.
- V$SESSTAT does not include the name of the statistic. In order to find the statistic name, this view must be joined to either V$SYSSTAT or V$STATNAME.

V$SESSTAT can be used to find sessions with the following:

- The highest resource usage
- The highest average resource usage rate (ratio of resource usage to logon time)
- The current resource usage rate (delta between two snapshots)

Useful Statistics in V$SESSTAT

The most referenced statistics in V$SESSTAT are a subset of those described for V$SYSSTAT and include `session logical reads`, `CPU used by this session`, `db block changes`, `redo size`, `physical writes`, `parse count (hard)`, `parse count (total)`, `sorts (memory)`, and `sorts (disk)`.

Useful Columns for V$SESSTAT

- SID: Session identifier
- STATISTIC#: Resource identifier
- VALUE: Resource usage

Join Columns for V$SESSTAT

Columns:	Fixed View:	Joined Column(s):
STATISTIC#	V$STATNAME	STATISTIC#
SID	V$SESSION	SID

Example - Finding the Top Sessions with Highest Logical and Physical I/O Rates Currently Connected to the Database

The following SQL statement shows the logical and physical read rates (per second) for all active sessions connected to the database. Rates for logical and physical I/O are calculated using the elapsed time since logon (from V$SESSION.LOGON_TIME). This might not be particularly accurate for sessions connected to the database for long periods, but it is sufficient for this example.

To determine the STATISTIC#'s for the `session logical reads` and `physical reads` statistics:

```
SELECT name, statistic#
  FROM V$STATNAME
 WHERE name IN ('session logical reads','physical reads') ;

NAME                           STATISTIC#
------------------------------ ----------
session logical reads                   9
physical reads                         40
```

Use these values in the following query, which orders the sessions by resource usage:

```
SELECT ses.sid
     , DECODE(ses.action,NULL,'online','batch')          "User"
     , MAX(DECODE(sta.statistic#,9,sta.value,0))
       /greatest(3600*24*(sysdate-ses.logon_time),1)     "Log IO/s"
     , MAX(DECODE(sta.statistic#,40,sta.value,0))
       /greatest(3600*24*(sysdate-ses.logon_time),1)     "Phy IO/s"
     , 60*24*(sysdate-ses.logon_time)                    "Minutes"
 FROM V$SESSION ses
    , V$SESSTAT sta
WHERE ses.status     = 'ACTIVE'
  AND sta.sid        = ses.sid
  AND sta.statistic# IN (9,40)
GROUP BY ses.sid, ses.action, ses.logon_time
ORDER BY
        SUM( DECODE(sta.statistic#,40,100*sta.value,sta.value) )
      / greatest(3600*24*(sysdate-ses.logon_time),1)  DESC;

  SID User   Log IO/s Phy IO/s Minutes
----- ------ -------- -------- -------
 1951 batch       291    257.3       1
  470 online    6,161     62.9       0
  730 batch     7,568     43.2     197
 2153 online    1,482     98.9      10
 2386 batch     7,620     35.6      35
 1815 batch     7,503     35.5      26
 1965 online    4,879     42.9      19
 1668 online    4,318     44.5       1
 1142 online      955     69.2      35
 1855 batch       573     70.5       8
 1971 online    1,138     56.6       1
 1323 online    3,263     32.4       5
 1479 batch     2,857     35.1       3
  421 online    1,322     46.8      15
 2405 online      258     50.4       8
```

To better show the impact of each individual session on the system, the results were ordered by the total resource usage per second. The resource usage was calculated by adding `session logical reads` and (a weighted) `physical reads`.

Physical reads was weighted by multiplying the raw value by a factor of 100, to indicate that a physical I/O is significantly more expensive than reading a buffer already in the cache.

To calculate the physical I/O weighting factor, the following assumptions were made:

- Average wait for a physical I/O (PIO) was 10 ms (queried from V$SYSTEM_EVENT.AVERAGE_WAIT for the events db file sequential read and db file scattered read).
- Average logical I/O rate (LIO) was 13000/second/CPU (queried from V$SYSSTAT for the statistic name session logical reads. This statistic was divided by the elapsed time in seconds and the number of CPUs on the system).
- This provides a ratio of 130 logical reads per 10 ms, and 1 physical read per 10 ms for this configuration. This ratio was rounded to the ballpark number of 100.

V$SQL

A SQL statement can map to multiple cursors, because the objects referred to in the cursor can differ from user to user. If there are multiple cursors (child cursors) present, then V$SQLAREA provides aggregated information for all the cursors.

For looking at individual cursors, V$SQL can be used. This view contains cursor level details for the SQL. It can be used when trying to locate the session or person responsible for parsing the cursor.

V$SQL_PLAN

This view provides a way of examining the execution plan for cursors that were recently executed.

The information in this view is very similar to the output of an EXPLAIN PLAN statement. However, EXPLAIN PLAN shows a theoretical plan that can be used if this statement were to be executed, whereas V$SQL_PLAN contains the actual plan used. The execution plan obtained by the EXPLAIN PLAN statement can be different from the execution plan used to execute the cursor, because the cursor might have been compiled with different values of session parameters (for example, HASH_AREA_SIZE).

Uses for V$SQL_PLAN Data

- Determining the current execution plan
- Identifying the effect of creating an index on a table
- Finding cursors containing a certain access path (for example, full table scan or index range scan)
- Identifying indexes that are, or are not, selected by the optimizer
- Determining whether the optimizer selects the particular execution plan (for example, nested loops join) expected by the developer

This view can also be used as a key mechanism in plan comparison. Plan comparison can be useful when the following types of changes occur:

- Dropping or creating indexes
- Running the ANALYZE statement on the database objects
- Modifying initialization parameter values
- Switching from the rule-based optimizer to the cost-based optimizer
- After migrating the application or the database to a new release

If previous plans are kept (for example, selected from V$SQL_PLAN and stored in permanent Oracle tables for reference), then it is then possible to identify how changes in the performance of a SQL statement can be correlated with changes in the execution plan for that statement.

Useful Columns for V$SQL_PLAN

The view contains almost all PLAN_TABLE columns, in addition to new columns. The columns that are also present in the PLAN_TABLE have the same values:

- ADDRESS: Address of the handle to the parent for this cursor
- HASH_VALUE: Hash value of the parent statement in the library cache

The two columns ADDRESS and HASH_VALUE can be used to join with V$SQLAREA to add the cursor-specific information.

- CHILD_NUMBER: Child cursor number using this execution plan

The columns ADDRESS, HASH_VALUE and CHILD_NUMBER can be used to join with V$SQL to add the child cursor specific information.

- OPERATION: Name of the internal operation performed in this step; for example, TABLE ACCESS
- OPTIONS: A variation on the operation described in the OPERATION column; for example, FULL
- OBJECT_NODE: 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
- OBJECT#: Object number of the table or the index
- OBJECT_OWNER: Name of the user who owns the schema containing the table or index
- OBJECT_NAME: Name of the table or index
- OPTIMIZER: Current mode of the optimizer for the first row in the plan (statement line); for example, CHOOSE. In case the operation is a database access (e.g, TABLE ACCESS), it tells whether the object is analyzed or not
- ID: A number assigned to each step in the execution plan
- PARENT_ID: ID of the next execution step that operates on the output of the current step
- DEPTH: The depth (or level) of the operation in the tree; that is, it is not necessary to do a CONNECT BY to get the level information generally used to indent the rows from the PLAN_TABLE - the root operation (statement) has level 0.
- POSITION: Order of processing for operations that all have the same PARENT_ID

- COST: 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
- CARDINALITY: The estimate, by the cost-based optimizer, of the number of rows produced by the operation
- BYTES: The estimate, by the cost-based optimizer, of the number of bytes produced by the operation
- OTHER_TAG: Describes the contents of the OTHER column (see Chapter 9, "Using EXPLAIN PLAN" for values)
- PARTITION_START: The start partition of a range of accessed partition
- PARTITION_STOP: The stop partition of a range of accessed partitions
- 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 (see Chapter 9, "Using EXPLAIN PLAN" for values)
- DISTRIBUTION: For parallel query, stores the method used to distribute rows from producer query servers to consumer query servers.
- CPU_COST: The CPU 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
- IO_COST: The I/O 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
- TEMP_SPACE: The temporary space usage of the operation (sort or hash-join) as estimated by the optimizer's cost-based approach; for statements that use the T

The DEPTH column replaces the LEVEL pseudo-column produced by the CONNECT BY operator, which sometimes is used in SQL scripts to help indent the PLAN_TABLE data.

Join Columns for V$SQL_PLAN

The columns ADDRESS, HASH_VALUE and CHILD_NUMBER are used to join with V$SQL or V$SQLAREA to fetch the cursor-specific information; for example, BUFFER_GETS, or with V$SQLTEXT to return the full text of the SQL statement.

Column(s):	Fixed View:	Joined Column(s)
address, hash_value	v$sqlarea	address, hash_value
address, hash_value, child_number	v$sql	address, hash_value child_number
address, hash_value	v$sqltext	address, hash_value

Example - Determining the Optimizer Plan for a SQL Statement

The following statement shows the EXPLAIN PLAN for a specified SQL statement. Looking at the plan for a SQL statement is one of the first steps in tuning a SQL statement. The SQL statement to return the plan for is identified by the statement's HASH_VALUE and address.

> **See Also:** "Identifying and Gathering Data on Resource-Intensive SQL" on page 6-3 for information on how to identify SQL statements to tune

A SQL*Plus query and sample output from V$SQL_PLAN (assumes only one child cursor):

```
column id          format 999 newline
column operation   format a20
column operation   format a20
column options     format a15
column object_name format a22 trunc
column optimizer   format a3  trunc
```

```
SELECT id
     , lpad (' ', depth) || operation operation
     , options
     , object_name
     , optimizer
     , cost
  FROM V$SQL_PLAN
 WHERE hash_value = 2446703096
   AND address    = '80E3A1D8'
 START WITH id = 0
 CONNECT BY
       (     prior id           = parent_id
         AND prior hash_value   = hash_value
         AND prior child_number = child_number
       )
 ORDER SIBLINGS BY id, position;

  ID OPERATION            OPTIONS         OBJECT_NAME            OPT       COST
---- -------------------- --------------- ---------------------- --- ----------
   0 SELECT STATEMENT                                            CHO
   1  SORT                ORDER BY                                          6
   2   NESTED LOOPS                                                         4
   3    HASH JOIN                                                           3
   8    TABLE ACCESS      BY INDEX ROWID  ORDERS                            1
   9     INDEX            RANGE SCAN      ORDERS_PK                         2
   4     TABLE ACCESS     BY INDEX ROWID  ORDERS_LINE_ITEM                  1
   5      INDEX           RANGE SCAN      ORDERS_LINE_ITEM_PK               2
   6     TABLE ACCESS     BY INDEX ROWID  ORDERS_STATUS                     1
   7      INDEX           RANGE SCAN      ORDERS_STATUS_PK                  2

10 rows selected.
```

V$SQL_PLAN shows the plan for a cursor, not for a SQL statement. The difference is that a SQL statement can have more than one cursor associated with it, with each cursor further identified by a CHILD_NUMBER. Below are a few examples of how a SQL statement can result in more than one cursor:

- When the same table name resolves to two separate tables:

  User1: SELECT * FROM EMPLOYEE;

  User2: SELECT * FROM EMPLOYEE;

  Where user2 has his own employee table, and user1 uses the table referenced by a public synonym.

- When the environment for user1 differs from user2. For example, if user2 specified the first rows (ALTER SESSION SET OPTIMIZER_GOAL = FIRST_ROWS) in their login script, and user1 did not.

  If the results of querying V$SQL_PLAN for a HASH_VALUE and ADDRESS result in more than one plan appearing, it is because this SQL statement has more than one child cursor. In this case, for each child cursor (identified by CHILD_NUMBER), look at the plan to identify whether they differ significantly.

V$SQLAREA

This view keeps track of all the shared cursors present in the shared pool. It has one row for every SQL statement present in the shared pool. It is an invaluable view for finding the resource usage of a SQL statement.

Information columns in V$SQLAREA

- HASH_VALUE: Hash value of the SQL statement
- ADDRESS: SGA address for the SQL statement

The above two columns are used to identify the SQL statement. Sometimes, two different statements could hash to the same value. In such cases, it is necessary to use the address along with the hash_value.

- PARSING_USER_ID: User who parsed the first cursor for the statement
- VERSION_COUNT: Number of cursors for the statement
- KEPT_VERSIONS: Cursors of the statement pinned using DBMS_SHARED_POOL.KEEP()
- SHARABLE_MEMORY: Total shared memory used by the cursor
- PERSISTENT_MEMORY: Total persistent memory used by the cursor
- RUNTIME_MEMORY: Total runtime memory used by the cursor
- SQL_TEXT: Up to first 1000 characters of SQL statement
- MODULE, ACTION: Information about the session parsing the first cursor if set using DBMS_APPLICATION_INFO

Other Useful Columns in V$SQLAREA

These columns get incremented with each execution of the statement.

- BUFFER_GETS: Logical I/Os caused by this statement
- DISK_READS: Physical I/O requests due to the statement
- SORTS: Sorts performed due to the statement
- CPU_TIME: Time taken by the CPU due to the statement
- ELAPSED_TIME: Total time spent due to the statement
- PARSE_CALLS: Parse calls (hard and soft) for the statement
- EXECUTIONS: Number of times the statement was executed

- INVALIDATIONS: Total invalidations for statement's cursors
- LOADS: Loads (and reloads) for the statement
- ROWS_PROCESSED: Total rows processed by this statement

Join Columns in V$SQLAREA

Columns:	Fixed View:	Joined Column(s):
HASH_VALUE, ADDRESS	V$SESSION	SQL_HASH_VALUE, SQL_ADDRESS
HASH_VALUE, ADDRESS	V$SQLTEXT, V$SQL, V$OPEN_CURSOR	HASH_VALUE, ADDRESS
SQL_TEXT	V$DB_OBJECT_CACHE	NAME

Example - Finding Resource-Intensive SQL

There are several costs you can use:

- Total logical I/O (LIO), LIO/execution
- Total physical I/O (PIO), PIO/execution
- PIO/LIO (poor cache hit ratio)
- parse_calls, parse_calls/executions

```
SELECT hash_value, executions, buffer_gets, disk_reads, parse_calls
  FROM V$SQLAREA
 WHERE buffer_gets > 10000000
    OR disk_reads > 1000000
 ORDER BY buffer_gets + 100*disk_reads DESC;
```

```
HASH_VALUE EXECUTIONS BUFFER_GETS DISK_READS PARSE_CALLS
---------- ---------- ----------- ---------- -----------
2676594883        126     7583140    6199113         126
4074144966        126     7264362    6195433          49
 228801498        136   236116544    2371187         136
 360282550       5467    21102603    4476317        2355
1559420740        201     8197831    4537591          39
3213702248   28039654   364516977         44         131
1547710012        865     7579025    3337735         865
3000880481       4481     3676546    2212658        2885
1398193708       4946    73018658    1515257        1418
1052917712    8342025   201246652      38240      327462
 371697988          7    74380777     862611           7
1514306888    3922461    29073852    1223482         268
1848522009          1     1492281    1483635           1
1478599096   28042103   140210513        594         164
 226079402      21473    22121577    1034787        4484
 478652562       4468    21669366    1020370        4438
2054874295      73520   118272694      29987       73520
```

Note: If a statement is executing for the first time on the system and responsible for large fraction of the current resource usage, then this statement does not find that statement, because the BUFFER_GETS and DISK_READS statistics do not get updated until the statement finishes execution.

Example - Finding Resources Used by a SQL Statement

```
SELECT hash_value, buffer_gets, disk_reads, executions, parse_calls
  FROM V$SQLAREA
 WHERE hash_Value = 228801498
   AND address = hextoraw('CBD8E4B0');

HASH_VALUE BUFFER_GETS DISK_READS EXECUTIONS PARSE_CALLS
---------- ----------- ---------- ---------- -----------
 228801498   236116544    2371187        136         136
```

V$SQLTEXT

This view contains the complete SQL text for the SQL statements in the shared pool.

Note: V$SQLAREA only contains only the first 1000 characters.

Useful Columns for V$SQLTEXT

- HASH_VALUE: Hash value for the SQL statement
- ADDRESS: Address of the SQL statement cursor in SGA
- SQL_TEXT: Statement text in 64 character chunks
- PIECE: Ordering information for the SQL statement pieces

Join Columns for V$SQLTEXT

Columns:	Fixed View:	Joined Column(s):
HASH_VALUE, ADDRESS	VSQL, VSESSION	HASH_VALUE, ADDRESS
HASH_VALUE. ADDRESS	V$SESSION	SQL_HASH_VALUE. SQL_ADDRESS

Example - Finding the SQL Statement for a Hash Value

```
SELECT sql_text
  FROM V$SQLTEXT
 WHERE hash_value = 228801498
 ORDER BY piece;

SQL_TEXT
----------------------------------------------------------------
select dbsu.primary_flag, i.site_use_code, i.rowid
from ra_customers dbc, ra_addresses dbad, ra_site_uses dbsu, ra_customers_
interface i
where (((((((i.orig_system_customer_ref=dbc.orig_system_reference and
dbad.address_id=dbsu.address_id) and i.site_use_code=dbsu.site_use_code) and
dbsu.status='A') and dbad.customer_id=dbc.customer_id) and i.request_id=:b0) and
nvl(i.validated_flag,'N')<>'Y') and ((i.primary_site_use_flag='Y' and
dbsu.primary_flag='Y') or dbsu.site_use_code in ('STMTS','DUN','LEGAL')))
group by dbsu.primary_flag,i.orig_system_customer_ref,i.site_use_code,i.insert_
update_flag,i.rowid
```

V$SYSSTAT

V$SYSSTAT stores instance-wide statistics on resource usage, cumulative since the instance was started.

Similar to V$SESSTAT, this view stores the following types of statistics:

- A count of the number of times an action occurred (`user commits`)
- A running total of volumes of data generated, accessed, or manipulated (`redo size`)
- If TIMED_STATISTICS is true, then the cumulative time spent performing some actions (`CPU used by this session`)

Useful Columns in V$SYSSTAT

- STATISTIC#: Identifier for the statistic
- NAME: Statistic name
- VALUE: Resource usage

The value for each statistic stores the resource usage for that statistic since instance startup. Below are sample column values for the statistic `execute count`.

Statistic#:	Name:	Value:
215	execute count	19,003,070

Note: The STATISTIC# for a statistic can change between releases. Do not rely on STATISTIC# to remain constant. Instead, use the statistic NAME column to query the VALUE.

Uses for V$SYSSTAT Data

The data in this view is used for monitoring system performance. Derived statistics, such as the buffer cache hit ratio and soft parse ratio, are computed from V$SYSSTAT data.

Data in this view is also used for monitoring system resource usage and how the system's resource usage changes over time. As with most performance data, examine the system's resource usage over an interval. To do this, take a snapshot of the data within the view at the beginning of the interval and another at the end. The difference in the values (end value - begin value) for each statistic is the resource

used during the interval. This is the methodology used by Oracle tools such as Statspack and BSTAT/ESTAT.

In order to compare one interval's data with another, the data can be normalized (per transaction, per execution, per second, or per logon). Normalizing the data on both workloads makes identifying the variances between the two workloads easier. This type of comparison is especially useful after patches have been applied, applications have been upgraded, or simply over time to see how increases in user population or data growth affects the resource usage.

You can also use V$SYSSTAT data to examine the resource consumption of contended-for resources that were identified by querying the V$SYSTEM_EVENT view.

Useful Statistics for V$SYSSTAT

Below are some of the V$SYSSTAT statistics that are most useful during tuning, along with an explanation of the statistic. This list is in alphabetical order.

> **See Also:** *Oracle9i Database Reference* for a complete list of statistics and their description

Key Database Usage Indicators

- CPU used by this session: The total amount of CPU used by all sessions, excluding background processes. This unit for this statistic is hundredths of a second. Calls that complete in less than 10ms are rounded up to this unit.
- db block changes: The number of changes made to database blocks in the SGA that were part of an insert, update, or delete operation. This statistic is a rough indication of total database work. On a per transaction level, this statistic indicates the rate at which buffers are being dirtied.
- execute count: The total number of SQL statement executions (including recursive SQL).
- logons current: Sessions currently connected to the instance. When using two snapshots across an interval, an average value (rather than the difference) should be used.
- logons cumulative: The total number of logons since the instance started. To determine the number of logons in a particular period, subtract the end value from the begin value. A useful derived statistic is to divide the number of connections between a begin and end time, and divide this by the number of seconds the interval covered. This gives the logons rate per second. Optimally,

there should be no more than two logons per second. To contrast, a logon rate of 50 per second is considered very high. Applications that continually connect and disconnect from the database (for example, once per transaction) do not scale well.

- `parse count (hard)`: The number of parse calls that resulted in a miss in the shared pool. A hard parse occurs when a SQL statement is executed and the SQL statement is either not in the shared pool, or it is in the shared pool but it cannot be shared because part of the metadata for the two SQL statements is different. This can happen if a SQL statement is textually identical to a preexisting SQL statement, but the tables referred to in the two statements resolve to physically different tables. A hard parse is a very expensive operation in terms of CPU and resource use (for example, latches), because it requires Oracle to allocate memory within the shared pool, then determine the execution plan before the statement can be executed.
- `parse count (total)`: The total number of parse calls, both hard and soft. A soft parse occurs when a session executes a SQL statement, and the statement is already in the shared pool and can be used. For a statement to be used (that is, shared) all data pertaining to the existing SQL statement (including data such as the optimizer execution plan) must be equally applicable to the current statement being issued. These two statistics are used to calculate the soft-parse ratio.
- `parse time cpu`: Total CPU time spent parsing in hundredths of a second. This includes both hard and soft parses.
- `parse time elapsed`: The total elapsed time for the parse call to complete.
- `physical reads`: The number of blocks read from the operating system. It includes physical reads into the SGA buffer cache (a buffer cache miss) and direct physical reads into the PGA (for example, during direct sort operations). This statistic is *not* the number of I/O requests.
- `physical writes`: The number of database blocks written from the SGA buffer cache to disk by DBWR and from the PGA by processes performing direct writes.
- `redo log space requests`: The number of times a server process waited for space in the redo logs, typically because a log switch is needed.
- `redo size`: The total amount of redo generated (and hence written to the log buffer), in bytes. This statistic (normalized per second or per transaction) is a good indicator of update activity.

- session logical reads: The number of logical read requests that can be satisfied in the buffer cache or by a physical read.
- sorts (memory) and sorts (disk): sorts (memory) is the number of sort operations that fit inside the SORT_AREA_SIZE (and hence did not require an on-disk sort). sorts (disk) is the number of sort operations that were larger than SORT_AREA_SIZE and had to use space on disk to complete the sort. These two statistics are used to compute the in-memory sort ratio.
- sorts (rows): The total number of rows sorted. This statistic can be divided by the 'sorts (total)' statistic to determine rows per sort. It is an indicator of data volumes and application characteristics.
- table fetch by rowid: The number of rows returned using ROWID (due to index access or because a SQL statement of the form "where rowid = &rowid" was issued).
- table scans (rows gotten): The total number of rows processed during full table scans.
- table scans (blocks gotten): The number of blocks scanned during full table scans, excluding those for split rows.
- user commits + user rollbacks: This provides the total number of transactions on the system. This number is used as the divisor when calculating the per-transaction ratios for other statistics. For example, to calculate the number of logical reads per transaction, use the following formula: session logical reads / (user commits + user rollbacks).

Notes on Physical I/O

A physical read as reported by Oracle might not result in an actual physical disk I/O operation. This is possible because most operating systems have an operating system filesystem cache where the block might be present. Alternatively, the block might also be present in disk or controller level cache, again avoiding an actual I/O. A physical read as reported by Oracle merely indicates that the required block was not in the buffer cache (or in the case of a direct read operation, was required to be read into private memory).

Instance Efficiency Ratios From V$SYSSTAT Statistics

Below are typical instance efficiency ratios calculated from V$SYSSTAT data. Each ratio's computed value should all be as close as possible to 1:

Buffer cache hit ratio: This is a good indicator of whether the buffer cache is too small.

```
1 - ((physical reads - physical reads direct - physical reads
direct (lob)) / session logical reads)
```

Soft parse ratio: This shows whether there are many hard parses on the system. The ratio should be compared to the raw statistics to ensure accuracy. For example, a soft parse ratio of .2 typically indicates a high hard parse rate. However, if the total number of parses is low, then the ratio should be disregarded.

```
1 - ( parse count (hard) / parse count (total) )
```

In-memory sort ratio: This shows the proportion of sorts that are performed in memory. Optimally, in an operational (OLTP) system, most sorts are small and can be performed solely as in-memory sorts.

```
sorts (memory) / ( sorts (memory) + sorts (disk) )
```

Parse to execute ratio: In an operational environment, optimally a SQL statement should be parsed once and executed many times.

```
1 - (parse count / execute count)
```

Parse CPU to total CPU ratio: This shows how much of the total CPU time used was spent on activities other than parsing. When this ratio is low, the system is performing too many parses.

```
1 - (parse time cpu / CPU used by this session)
```

Parse time CPU to parse time elapsed: Often, this can indicate latch contention. The ratio calculates whether the time spent parsing is allocated to CPU cycles (that is, productive work) or whether the time spent parsing was not spent on CPU cycles. Time spent parsing not on CPU cycles usually indicates that the time was spent sleeping due to latch contention.

```
parse time cpu / parse time elapsed
```

Load Profile Data from V$SYSSTAT Statistics

To determine the load profile of the system, normalize the following statistics per second and per transaction: `logons cumulative`, `parse count (total)`, `parse count (hard)`, `executes`, `physical reads`, `physical writes`, `block changes`, and `redo size`.

The normalized data can be examined to see if the 'rates' are high, or it can be compared to another baseline data set to identify how the system profile is changing over time. For example, block changes per transaction is calculated by the following:

```
db block changes / ( user commits + user rollbacks )
```

Additional computed statistics that measure load include the following:

Blocks changed per read: This shows the proportion of block changes to block reads. It is an indication of whether the system is predominantly read only or whether the system performs many data changes (inserts / updates / deletes).

```
db block changes / session logical reads
```

Rows per sort:

```
sorts (rows) / ( sorts (memory) + sorts (disk) )
```

Join Columns for V$SYSSTAT

Column:	Fixed View:	Joined Column(s):
STATISTIC#	V$STATNAME	STATISTIC#

V$SYSTEM_EVENT

This view is a summary of waits for an event by an instance. While V$SESSION_WAIT shows the current waits on the system, V$SYSTEM_EVENT provides a summary of all the event waits on the instance since it started. It is useful to get a historical picture of waits on the system. By taking two snapshots and doing the delta on the waits, you can determine the waits on the system in a given time interval.

Useful Columns for V$SYSTEM_EVENT

- EVENT: Name of the wait event
- TOTAL_WAITS: Total number of waits for this event
- TIME_WAITED: Total time waited for this event (in hundredths of a second)
- AVERAGE_WAIT: Average amount of time waited for this event by this session (in hundredths of a second)
- TOTAL_TIMEOUTS: Number of times the wait timed out

Example - Finding the Total Waits on the System

```
SELECT event, total_waits waits, total_timeouts timeouts,
       time_waited total_time, average_wait avg
  FROM V$SYSTEM_EVENT
  ORDER BY 4 DESC;

EVENT                                WAITS  TIMEOUTS TOTAL_TIME      AVG
---------------------------------- ---------- --------- ---------- --------
SQL*Net message from client        112079628         0 8622695365    76.93
virtual circuit status              83559794   1168000 4275791401    51.17
rdbms ipc message                  131463191 115900505 2865926648    21.80
dispatcher timer                   311975975 168152330 2296760866     7.36
PX Idle Wait                         7198490   7198559 1439690729   199.99
pmon timer                            939711    939639  287866277   306.33
smon timer                              9892      9114  287627013 29076.73
lock manager wait for remote mes     72001548  71967858  287526387     3.99
db file sequential read              29419894         0   32395392     1.10
PL/SQL lock timer                       19725     19688   29702609  1505.83
log file sync                         7055611        86    9550819     1.35
log file parallel write               7184801         4    8123534     1.13
SQL*Net more data from client          991402         0    3543149     3.57
db file parallel write                 727317         0    3012928     4.14
control file parallel write            950531         0    1975646     2.07
```

log file sequential read	1162465	0	813715	0.69
enqueue	9975	7692	423191	42.42
direct path read	453873	0	298944	0.65
db file scattered read	347172	0	292875	0.84
row cache lock	472207	25	169365	0.35
direct path write	124323	0	132075	1.06
buffer busy due to global cache	148122	0	122381	0.82
SQL*Net more data to client	17171954	52	101762	0.00
db file parallel read	68849	0	100842	1.46
DFS lock handle	18615	1080	97651	5.24
SQL*Net message to client	112079756	0	77604	0.00
control file sequential read	65793	0	62560	0.95
buffer busy waits	132402	97	60351	0.45
latch free	67675	57975	58365	0.86
log file switch completion	1449	24	34244	23.63
db file single write	10868	0	25518	2.34
SQL*Net break/reset to client	19130	0	9387	0.49
LGWR wait for redo copy	120199	356	8613	0.07
global cache lock busy	4447	0	7574	1.70
undo segment extension	5363841	5363828	6375	0.00
log file single write	2143	0	6267	2.92
refresh controlfile command	2644	0	4837	1.82
library cache load lock	49	10	3859	78.75
file open	178566	0	2930	0.01
switch logfile command	100	0	2468	24.68
library cache pin	9261	1	1716	0.18
pipe get	9	3	1460	162.22
rdbms ipc reply	10296	0	846	0.08
wait for gms registration	32	32	672	21.00
process startup	43	2	662	15.39
file identify	5438	0	584	0.10
control file single write	332	0	475	1.43
Null event	17	17	409	24.05
log buffer space	18	0	209	11.61
wait for lock db to unfreeze	1	1	199	199.00
local write wait	11	0	44	4.00
LMON wait for LMD to inherit commu	1	1	10	10.00
wait for lock db to become frozen	2	2	3	1.50
instance state change	2	0	0	0.00
global cache bg acks	2	0	0	0.00
buffer deadlock	141	141	0	0.00

To find the bottlenecks:

- Remove the idle events.
- Examine the time spent waiting for different events.
- Examine the average time per wait also, because some waits (like log file switch completion) might happen only periodically, but cause a big performance hit when they happen.

V$UNDOSTAT

This view monitors how undo space and transactions are executed in the current instance. Statistics for undo space consumption, transaction concurrency, and length of queries in the instance are available.

Useful Columns for V$UNDOSTAT

- `Endtime`: End time for each ten minute interval
- `UndoBlocksUsed`: Total number of undo blocks consumed
- `TxnConcurrency`: Maximum number of transactions executed concurrently
- `TxnTotal`: Total number of transactions executed within the interval
- `QueryLength`: Maximum length of queries, in seconds executed in the instance
- `ExtentsStolen`: Number of times an undo extent must be transferred from one undo segment to another within the interval
- `SSTooOldError`: Number of 'Snapshot Too Old' errors that occurred within the interval
- `UNDOTSN`: undo tablespaces in service during each time period

The first row of the view shows statistics for the current time interval. Each subsequent row represents a ten minute interval. There is a total of 144 rows, spanning a 24 hour cycle.

Example - V$UNDOSTAT

The example below shows how undo space is consumed in the system for the previous 24 hours from the time 16:07.

```
SELECT * FROM V$UNDOSTAT;

End-Time UndoBlocks  TxnConcrcy  TxnTotal  QueryLen ExtentsStolen  SSTooOldError
-------- ----------  ----------  --------  -------- -------------  -------------
16:07          252          15      1511     25         2              0
16:00          752          16      1467    150         0              0
15:50          873          21      1954     45         4              0
15:40         1187          45      3210    633        20              1
15:30         1120          28      2498   1202         5              0
15:20          882          22      2002     55         0              0
```

Among the statistics collected, you see that the peak undo consumption happened at the interval of (15:30, 15:40). 1187 undo blocks were consumed in 10 minutes (or about two blocks per second). Also, the highest transaction concurrency occurred during that same period with 45 transactions executing at the same time. The longest query (1202 seconds) was executed (and ended) in the period (15:20, 15:30). Note that the query actually was started in the interval (15:00, 15:10) and continued until around 15:20.

V$WAITSTAT

This view keeps a summary all buffer waits since instance startup. It is useful for breaking down the waits by class if you see a large number of buffer busy waits on the system.

Useful Columns for V$WAITSTAT

- `class`: Class of block (data segment header, undo segment header, data block)
- `waits`: Number of waits for this class of blocks
- `time`: Total time waited for this class of block

Reasons for Waits

The following are possible reasons for waits:

- Undo segment header: not enough rollback segments
- Data segment header / freelist: freelist contention
- Data block
- Large number of CR clones for the buffer
- Range scans on indexes with large number of deletions
- Full table scans on tables with large number of deleted rows
- Blocks with high concurrency

> **See Also:** Chapter 22, "Instance Tuning" for more information on wait events

A

Schemas Used in Performance Examples

The following tables are used in various examples in this book. The statistics are from representative systems.

Note: These schemas are used in examples in Chapter 1, "Introduction to the Optimizer" and Chapter 9, "Using EXPLAIN PLAN".

PER_ALL_PEOPLE_F

This table stores data for employees on the system. For large corporations, it is common to have 10,000 to 30,000 rows in this table. The unique key is a concatenated index, but `person_id` by itself is quite selective also. Other selective columns are `employee_number` and `full_name`.

Indexes on the Table

Unique	Index Name	Column Name
NO	PER_PEOPLE_F_FK1	BUSINESS_GROUP_ID
NO	PER_PEOPLE_F_FK2	PERSON_TYPE_ID
NO	PER_PEOPLE_F_N50	LAST_NAME
NO	PER_PEOPLE_F_N51	EMPLOYEE_NUMBER
NO	PER_PEOPLE_F_N52	APPLICANT_NUMBER
NO	PER_PEOPLE_F_N53	NATIONAL_IDENTIFIER
NO	PER_PEOPLE_F_N54	FULL_NAME
YES	PER_PEOPLE_F_PK	PERSON_ID
		EFFECTIVE_START_DATE
		EFFECTIVE_END_DATE

RA_CUSTOMERS

This table has a row for every customer in the system. For large companies, this table has several hundred thousand rows. The primary key is `customer_id`. Other selective columns are the following:

- `Customer_number`
- `Customer_name`
- `Orig_system_reference` (tracks the customer identifier for customers imported from another system)

Indexes on the Table

Unique	Index Name	Column Name
NO	RA_CUSTOMERS_N1	CUSTOMER_NAME
NO	RA_CUSTOMERS_N2	CREATION_DATE
NO	RA_CUSTOMERS_N3	CUSTOMER_KEY
NO	RA_CUSTOMERS_N4	JGZZ_FISCAL_CODE
YES	RA_CUSTOMERS_U1	**CUSTOMER_ID**
YES	RA_CUSTOMERS_U2	ORIG_SYSTEM_REFERENCE
YES	RA_CUSTOMERS_U3	CUSTOMER_NUMBER

SO_HEADERS_ALL / SO_HEADERS

This table has a row for every order on the system. For large companies, it is common to have several million rows in this table. The primary key is `header_id`, and there is another unique key on (`order_number`, `order_type_id`). Other selective columns are the following:

- `customer_id` (the customer placing the order)
- `purchase_order_num` (the purchase order for billing)
- `original_system_reference` (tracks the order identifier for orders imported from other systems)

Indexes on the Table

Unique	Index Name	Column Name
NO	SO_HEADERS_N1	CUSTOMER_ID
NO	SO_HEADERS_N10	WH_UPDATE_DATE
NO	SO_HEADERS_N2	OPEN_FLAG
NO	SO_HEADERS_N3	PURCHASE_ORDER_NUM
NO	SO_HEADERS_N4	INVOICE_TO_SITE_USE_ID
NO	SO_HEADERS_N5	ORIGINAL_SYSTEM_REFERENCE
NO	SO_HEADERS_N6	S1
NO	SO_HEADERS_N7	S4
NO	SO_HEADERS_N8	S6
NO	SO_HEADERS_N9	ORIGINAL_SYSTEM_REFERENCE
		ORIGINAL_SYSTEM_SOURCE_CODE
YES	SO_HEADERS_U1	**HEADER_ID**
YES	SO_HEADERS_U2	ORDER_NUMBER
		ORDER_TYPE_ID

MTL_SYSTEM_ITEMS

This table is the parts master for `so_lines_all`. It has a row for every part in every organization. The primary key is `inventory_item_id`, `organization_id`.

Indexes on the Table

Unique	Index Name	Column Name
NO	MTL_SYSTEM_ITEMS_N1	**ORGANIZATION_ID**
		SEGMENT1
NO	MTL_SYSTEM_ITEMS_N2	**ORGANIZATION_ID**
		DESCRIPTION
NO	MTL_SYSTEM_ITEMS_N3	INVENTORY_ITEM_STATUS_CODE
NO	MTL_SYSTEM_ITEMS_N4	**ORGANIZATION_ID**
		AUTO_CREATED_CONFIG_FLAG
NO	MTL_SYSTEM_ITEMS_N5	WH_UPDATE_DATE
NO	MTL_SYSTEM_ITEMS_N6	ITEM_CATALOG_GROUP_ID
		CATALOG_STATUS_FLAG
NO	MTL_SYSTEM_ITEMS_N7	PRODUCT_FAMILY_ITEM_ID
		ORGANIZATION_ID
NO	MTL_SYSTEM_ITEMS_N8	SEGMENT1
		SEGMENT2
		SEGMENT3
YES	MTL_SYSTEM_ITEMS_U1	**INVENTORY_ITEM_ID**
		ORGANIZATION_ID

SO_LINES_ALL / SO_LINES

This table has a row for every order line on the system. It joins to the `so_headers_all` table using `header_id`. Because an order has 10 to 12 lines, this table is 10 to 12 times the rows in `so_headers_all`. The primary key is `line_id`. Some other selective columns are the following:

- `header_id`
- `parent_line_id`
- `service_parent_line_id`
- `original_system_reference`

Indexes on the Table

Unique	Index Name	Column Name
NO	SO_LINES_N1	HEADER_ID
NO	SO_LINES_N10	S5
NO	SO_LINES_N11	S6
NO	SO_LINES_N12	S8
NO	SO_LINES_N13	S9
NO	SO_LINES_N14	S28
NO	SO_LINES_N15	S29
NO	SO_LINES_N16	S30
NO	SO_LINES_N17	PARENT_LINE_ID
NO	SO_LINES_N18	SHIPMENT_SCHEDULE_LINE_ID
NO	SO_LINES_N19	ATO_LINE_ID
NO	SO_LINES_N2	LINK_TO_LINE_ID
NO	SO_LINES_N20	SERVICE_PARENT_LINE_ID
NO	SO_LINES_N21	SHIP_TO_SITE_USE_ID
NO	SO_LINES_N22	SOURCE_LINE_ID
NO	SO_LINES_N23	ORIGINAL_SYSTEM_LINE_REFERENCE
NO	SO_LINES_N24	RETURN_REFERENCE_ID
NO	SO_LINES_N25	S27
NO	SO_LINES_N26	CREDIT_INVOICE_LINE_ID
NO	SO_LINES_N27	S25
NO	SO_LINES_N28	WH_UPDATE_DATE
NO	SO_LINES_N29	DEMAND_STREAM_ID
NO	SO_LINES_N3	OPEN_FLAG
NO	SO_LINES_N4	COMMITMENT_ID
NO	SO_LINES_N5	INVENTORY_ITEM_ID
NO	SO_LINES_N6	REQUEST_ID
NO	SO_LINES_N7	S2
NO	SO_LINES_N8	S3
NO	SO_LINES_N9	S4
YES	SO_LINES_U1	**LINE_ID**

Glossary

asynchronous I/O

Independent I/O, in which there is no timing requirement for transmission, and other processes can be started before the transmission has finished.

Autotrace

Generates a report on the execution path used by the SQL optimizer and the statement execution statistics. The report is useful to monitor and tune the performance of DML statements.

bind variable

A variable in a SQL statement that must be replaced with a valid value, or the address of a value, in order for the statement to successfully execute.

block

A unit of data transfer between main memory and disk. Many blocks from one section of memory address space form a segment.

bottleneck

The delay in transmission of data, typically when a system's bandwidth cannot support the amount of information being relayed at the speed it is being processed. There are, however, many factors that can create a bottleneck in a system.

buffer

A main memory address in which the buffer manager caches currently and recently used data read from disk. Over time, a buffer can hold different blocks. When a new block is needed, the buffer manager can discard an old block and replace it with a new one.

buffer pool

A collection of buffers.

cache

Also known as buffer cache. All buffers and buffer pools.

cache recovery

The part of instance recovery where Oracle applies all committed and uncommitted changes in the redo log files to the affected data blocks. Also known as the *rolling forward* phase of instance recovery.

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.

CBO

Cost-based optimizer. Generates a set of potential execution plans for SQL statements, estimates the cost of each plan, calls the plan generator to generate the plan, compares the costs, and chooses the plan with the lowest cost. This approach is used when the data dictionary has statistics for at least one of the tables accessed by the SQL statements. The CBO is made up of the query transformer, the estimator, and the plan generator.

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*.

contention

When some process has to wait for a resource that is being used by another process.

dictionary cache

A collection of database tables and views containing reference information about the database, its structures, and its users. Oracle accesses the data dictionary

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

distributed statement

A statement that accesses data on two or more distinct nodes/instances of a distributed database. A *remote statement* accesses data on one remote node of a distributed database.

dynamic performance views

The views database administrators create on dynamic performance tables (virtual tables that record current database activity). Dynamic performance views are called fixed views because they cannot be altered or removed by the database administrator.

enqueue

This is another term for a lock.

equijoin

A join condition containing an equality operator.

estimator

Uses statistics to estimate the selectivity, cardinality, and cost of execution plans. The main goal of the estimator is to estimate the overall cost of an execution plan.

EXPLAIN PLAN

A SQL statement that allows you to examine the execution plan chosen by the optimizer for DML statements. `EXPLAIN PLAN` causes the optimizer to choose an execution plan and then to put data describing the plan into a database table.

instance recovery

The automatic application of redo log records to Oracle uncommitted data blocks after a crash or system failure.

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.

latch

A simple, low-level serialization mechanism to protect shared data structures in the System Global Area.

library cache

A memory structure containing shared SQL and PL/SQL areas. The library cache is one of three parts of the shared pool.

LIO

Logical I/O. A block read which may or may not be satisfied from the buffer cache.

literal

A constant value, written at compile-time and read-only at run-time. Literals can be accessed quickly, and are used when modification is not necessary.

mirroring

Maintaining identical copies of data on one or more disks. Typically, mirroring is performed on duplicate hard disks at the operating system level, so that if one of the disks becomes unavailable, the other disk can continue to service requests without interruptions.

nonequijoin

A join condition containing something other than an equality operator.

optimizer

Determines the most efficient way to execute SQL statements by evaluating expressions and translating them into equivalent, quicker expressions. The optimizer formulates a set of execution plans and picks the best one for a SQL statement. See CBO.

Oracle Trace

Used by the Oracle Server to collect performance and resource utilization data, such as SQL parse, execute, fetch statistics, and wait statistics. Oracle Trace provides several SQL scripts that can be used to access server event tables, collects server event data and stores it in memory, and allows data to be formatted while a collection is occurring.

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.

paging

A technique for increasing the memory space available by moving infrequently-used parts of a program's working memory from main memory to a secondary storage medium, usually a disk. The unit of transfer is called a page.

parse

A *hard parse* occurs when a SQL statement is executed, and the SQL statement is either not in the shared pool, or it is in the shared pool but it cannot be shared. A SQL statement is not shared if the metadata for the two SQL statements is different. This can happen if a SQL statement is textually identical as a preexisting SQL statement, but the tables referred to in the two statements resolve to physically different tables, or if the optimizer environment is different.

A *soft parse* occurs when a session attempts to execute a SQL statement, and the statement is already in the shared pool, and it can be used (that is, shared). For a statement to be shared, all data, (including metadata, such as the optimizer execution plan) pertaining to the existing SQL statement must be equally applicable to the current statement being issued.

parse call

A call to Oracle to prepare a SQL statement for execution. This includes syntactically checking the SQL statement, optimizing it, and building (or locating) an executable form of that statement.

parser

Performs syntax analysis and semantic analysis of SQL statements, and expands views (referenced in a query) into separate query blocks.

PGA

Program Global Area. A nonshared memory region that contains data and control information for a server process, created when the server process is started.

PIO

Physical I/O. A block read which could not be satisfied from the buffer cache, either because the block was not present or because the I/O is a direct I/O (and bypasses the buffer cache).

plan generator

Tries out different possible plans for a given query so that the CBO can choose the plan with the lowest cost. It explores different plans for a query block by trying out different access paths, join methods, and join orders.

predicate

A `WHERE` condition in SQL.

query transformer

Decides whether to rewrite a user query to generate a better query plan, merges views, and performs subquery unnesting.

RAID

Redundant arrays of inexpensive disks. RAID configurations provide improved data reliability with the option of striping (manually distributing data). Different RAID configurations (levels) are chosen based on performance and cost, and are suited to different types of applications, depending on their I/O characteristics.

RBO

Rule-based optimizer. Chooses an execution plan for SQL statements based on the access paths available and the ranks of these access paths (if there is more than one way, then the RBO uses the operation with the lowest rank). The RBO is used if no statistics are available, otherwise the CBO is used.

row source generator

Receives the optimal plan from the optimizer and outputs the execution plan for the SQL statement. A row source is an iterative control structure that processes a set of rows in an iterated manner and produces a row set.

segment

A set of extents allocated for a specific type of database object such as a table, index, or cluster.

simple statement

An INSERT, UPDATE, DELETE, or SELECT statement that involves only a single table.

simple query

A SELECT statement that references only one table and does not make reference to GROUP BY functions.

SGA

System Global Area. A memory region within main memory used to store data for fast access. Oracle uses the shared pool to allocate SGA memory for shared SQL and PL/SQL procedures.

SQL Compiler

Compiles SQL statements into a shared cursor. The SQL Compiler is made up of the parser, the optimizer, and the row source generator.

SQL statements (identical)

Textually identical SQL statements do not differ in any way.

SQL statements (similar)

Similar SQL statements differ only due to changing literal values. If the literal values were replaced with bind variables, then the SQL statements would be textually identical.

SQL Trace

A basic performance diagnostic tool to help monitor and tune applications running against the Oracle server. SQL Trace lets you assess the efficiency of the SQL statements an application runs and generates statistics for each statement. The trace files produced by this tool are used as input for TKPROF.

SQL*Loader

Reads and interprets input files. It is the most efficient way to load large amounts of data.

Statspack

A set of SQL, PL/SQL, and SQL*Plus scripts that allow the collection, automation, storage, and viewing of performance data. Statspack supersedes the traditional UTLBSTAT/UTLESTAT tuning scripts.

striping

The interleaving of a related block of data across disks. Proper striping reduces I/O and improves performance.

TKPROF

A diagnostic tool to help monitor and tune applications running against the Oracle Server. TKPROF primarily processes SQL trace output files and translates them into readable output files, providing a summary of user-level statements and recursive SQL calls for the trace files. It can also assess the efficiency of SQL statements, generate execution plans, and create SQL scripts to store statistics in the database.

transaction recovery

The part of instance recovery where Oracle applies the rollback segments to undo the uncommitted changes. Also known as the *rolling back* phase of instance recovery.

UGA

User Global Area. A memory region in the large pool used for user sessions.

wait events

Statistics that are incremented by a server process/thread to indicate that it had to wait for an event to complete before being able to continue processing. Wait events are one of the first places for investigation when performing reactive performance tuning.

wait events (idle)

These events indicate that the server process is waiting because it has no work. These events should be ignored when tuning, because they do not indicate the nature of the performance bottleneck.

Index

A

access paths
  cluster join, 8-6
  cluster scans, 1-39
  composite index, 8-8
  defined, 1-8
  execution plans, 1-5
  hash cluster key, 8-6
  hash scans, 1-39
  index scans, 1-29
  indexed cluster key, 8-7
  single row by cluster join, 8-4
  single row by hash cluster key (with unique key), 8-4
  single row by rowid, 8-3
  single row by unique or primary key, 8-5
ALL operator, 2-22
ALL_OBJECTS view, 14-14
ALL_ROWS hint, 1-12, 5-6
allocation
  of memory, 14-2
ALTER INDEX statement, 4-7
ALTER SESSION statement
  examples, 10-5
  OPTIMIZER_GOAL parameter, 1-12
  SET SESSION_CACHED_CURSORS clause, 14-38
ALTER SYSTEM statement
  DISPATCHERS initialization parameter, 19-4
ANALYZE statement, 22-18
  creating histograms, 3-21
AND_EQUAL hint, 4-6, 5-17
anti-joins, 1-64
ANY operator, 2-21
APPEND hint, 5-33
applications
  data warehousing
    star queries, 1-65
ApplReg event, 12-15
array interface, 23-13
automatic segment-space management, 13-7, 15-20, 22-23
automatic undo management, 18-2
AUTOTRACE variable, 11-2

B

BEGIN_SNAP variable, 21-11
BETWEEN comparison operator, 2-22
binary files
  formatting using Oracle Trace, 12-3
bind variables, 14-21
  optimization, 1-41
BITMAP CONVERSION row source, 4-18
bitmap indexes, 4-12, 4-16
  inlist iterator, 9-19
  maintenance, 4-13
  on index-organized tables, 4-15
  vs. B-tree indexes, 4-12
  when to use, 4-11
bitmap join indexes, 4-18
BITMAP_MERGE_AREA_SIZE initialization parameter, 4-13, 4-17
bitmaps
  mapping to rowids, 4-15
block sampling, 3-4
bottlenecks

disk I/O, 15-3
memory, 14-2
network, 22-21
resource, 22-22
broadcast
distribution value, 9-24
B-tree indexes, 4-14, 4-17
buffer busy wait events, 22-22
actions, 22-23
buffer caches
reducing buffers, 14-11, 14-33
buffer pools
default cache, 14-12
KEEP cache, 14-12
multiple, 14-11
RECYCLE cache, 14-12
BYTES column
PLAN_TABLE table, 9-21

C

CACHE hint, 5-34
CARDINALITY column
PLAN_TABLE table, 9-21
CATALOG.SQL script, 13-4
CATPROC.SQL script, 13-4
chained rows, 22-18
CHAR datatype, 13-4
character set database option, 13-4
checkpoints
choosing checkpoint frequency, 17-3
CHOOSE hint, 1-12, 5-8
client/server applications, 16-10
CLUSTER hint, 5-11
clusters, 4-19
hash
scans of, 1-39, 8-4, 8-6
index
scans of, 8-7
joins and, 8-4, 8-6
scans of, 1-39, 8-4
hash, 8-4, 8-6
joins, 8-6
collections, 12-8
columns
pseudocolumns
ROWNUM, 2-34, 2-43, 8-15
selectivity, 3-2
histograms, 3-20
to index, 4-3
complex view merging, 2-34
composite indexes, 4-4
composite partitioning
examples of, 9-13
CONNECT BY clause
optimizing view queries, 2-34
Connection event, 12-15
connection manager, 23-14
connection pooling, 19-4
consistency
read, 22-17
consistent gets statistic, 14-8, 18-2
consistent mode
TKPROF, 10-13
constants
comparisons and, 2-18
evaluation of expressions, 2-18
when computed, 2-18
constraints, 4-8
contention
disk, 15-3
memory, 14-2, 22-1
tuning, 22-1
wait events, 22-36
context switches, 16-11
CONTROL_FILES initialization parameter, 13-10
COST column
PLAN_TABLE table, 9-21
cost-based optimizations, 1-15
extensible optimization, 1-72
histograms, 3-20
procedures for plan stability, 7-11
selectivity of predicates, 3-2
histograms, 3-20
user-defined, 1-73
star queries, 1-65
statistics, 3-2
user-defined, 1-72
upgrading to, 7-12
user-defined costs, 1-73

counter/accumulator views, 24-2
CPU_COUNT initialization parameter, 17-16
CPUs
  utilization, 16-9
CREATE DATABASE statement, 13-3
CREATE INDEX statement
  example, 14-59
  NOSORT clause, 14-59
  PARALLEL clause, 13-8
CREATE OUTLINE statement, 7-5
CREATE_BITMAP_AREA_SIZE initialization parameter, 4-13, 4-16
CREATE_STORED_OUTLINES parameter, 7-4
creating databases, 13-2
  manually, 13-2
  parameters, 13-2
  with Installer, 13-2
cross-facility 3 event, 12-18
cross-product items
  See also cross-facility 3 event, 12-17
current mode
  TKPROF, 10-13
current state views, 24-2
CURSOR_NUM column
  TKPROF_TABLE table, 10-18
CURSOR_SHARING initialization parameter, 1-68, 14-23, 14-43
CURSOR_SHARING_EXACT hint, 5-37
CURSOR_SPACE_FOR_TIME initialization parameter
  setting, 14-37

D

data cache, 16-2
data dictionary, 14-32
  scripts, 13-4
    CATALOG.SQL, 13-4
    CATPROC.SQL, 13-4
  statistics in, 3-15
  views used in optimization, 3-15
data indexing, 13-8
data loading, 13-8
Data Viewer
  tips on using
    collect data for specific wait events, 12-35
data warehousing
  dimensions, 1-65
  star queries, 1-65
database
  buffers, 14-11, 14-33
Database Connection event, 12-2
database creation
  manual, 13-2
  parameters, 13-2
  with Installer, 13-2
database identifier (DBID), 21-3
database options, 13-3
  character set, 13-4
  location of initial datafile, 13-4
  national character set, 13-4
  SQL.BSQ file, 13-4
Database Resource Manager, 16-4, 16-5, 16-9, 22-7
databases
  creating, 13-2
  distributed
    statement optimization on, 2-12
datatypes
  CHAR, 13-4
  NCHAR, 13-4
  NVARCHAR, 13-4
  NVARCHAR2, 13-4
  user-defined
    statistics, 1-72
  VARCHAR, 13-4
  VARCHAR2, 13-4
DATE_OF_INSERT column
  TKPROF_TABLE table, 10-18
db block gets statistic, 14-8, 18-2
DB file scattered read wait events, 22-25
  actions, 22-25
DB file sequential/scattered read wait events, 22-25, 22-27
DB_BLOCK_BUFFERS initialization parameter, 13-10, 14-11, 14-33
DB_BLOCK_SIZE initialization parameter, 13-3, 13-10, 15-12
DB_CACHE_ADVICE parameter, 14-6, 14-10
DB_CACHE_SIZE initialization parameter, 14-12
DB_DOMAIN initialization parameter, 13-10

DB_FILE_MULTIBLOCK_READ_COUNT
initialization parameter, 1-69, 13-7, 15-11, 15-12, 15-13, 22-25
cost-based optimization, 1-64
DB_NAME initialization parameter, 13-10
DB_nK_CACHE_SIZE initialization parameter, 14-11
DB_WRITER_PROCESSES initialization parameter, 22-35
DBA_OBJECTS view, 14-14
DBID (database identifier), 21-3
DBMS_JOB procedure, 21-7
DBMS_JOB.INTERVAL procedure, 21-8
DBMS_OUTLN package, 7-4
DBMS_OUTLN_EDIT package, 7-4
DBMS_SHARED_POOL package, 14-41
DBMS_STATS package, 3-5
creating histograms, 3-21
default cache, 14-12
DEFAULT_TABLESPACE variable, 21-6
deleting data, 21-20
deleting snapshots, 21-20
DEPTH column
TKPROF_TABLE table, 10-18
deterministic functions, 2-26
dictionary managed tablespaces, 21-4
dimensions
star joins, 1-65
star queries, 1-65
direct path read events, 22-28
actions, 22-29
causes, 22-29
direct path wait events, 22-30
direct path write events
actions, 22-30
causes, 22-30
direct-path INSERT, 5-33
disabled constraints, 4-8
Disconnect event, 12-15
disks
contention, 15-3
monitoring OS file activity, 22-8
dispatcher processes, 19-4
DISPATCHERS initialization parameter, 19-4, 23-3
DISTINCT operator
optimizing views, 2-34
distributed databases
statement optimization on, 2-12
distributed transactions
optimizing, 2-12
sample table scan not supported, 1-27
distribution
hints for, 5-29
DISTRIBUTION column
PLAN_TABLE table, 9-22
DML locks, 24-18
domain indexes
and EXPLAIN PLAN, 9-19
extensible optimization, 1-72
user-defined statistics, 1-72
using, 4-18
DRIVING_SITE hint, 5-25
duration events
in Oracle Trace, 12-2, 12-15
dynamic performance views
enabling statistics, 10-4

E

enabled constraints, 4-8
END_SNAP variable, 21-11
enforced constraints, 4-8
enqueue wait events
actions, 22-31
EPC_ERROR.LOG file, 12-36
equijoins, 6-8
ErrorStack event, 12-15
event timings, 21-19
examples
ALTER SESSION statement, 10-5
concurrently creating tablespaces, 13-6
CREATE DATABASE script, 13-4
CREATE INDEX statement, 14-59
creating indexes efficiently, 13-9
executing required data dictionary scripts, 13-5
execution plan, 8-20
EXPLAIN PLAN output, 8-20, 10-16
full table scan, 8-21
indexed query, 8-21
minimal initialization file, 13-11

NOSORT clause, 14-59
SET TRANSACTION statement, 18-3
SQL trace facility output, 10-16
V$DB_OBJECT_CACHE view, 24-4
V$FILESTAT view, 24-7
V$LATCH view, 24-10
V$LATCH_CHILDREN view, 24-12
V$LATCH_HOLDER view, 24-13
V$LIBRARYCACHE view, 24-15
V$LOCK view, 24-19, 24-20
V$OPEN_CURSOR view, 24-23, 24-24
V$PROCESS view, 24-28
V$ROLLSTAT view, 24-29
V$SESSION view, 24-34
V$SESSION_EVENT view, 24-36
V$SESSION_WAIT view, 24-39
V$SQLAREA view, 24-54, 24-55
V$SQLTEXT view, 24-56
V$SYSTEM_EVENT view, 24-63
Execute event, 12-15
execution plans
accessing views, 2-37, 2-39, 2-41
complex statements, 2-32
compound queries, 2-46, 2-47, 2-48
examples, 2-32, 8-20, 10-7
execution sequence of, 1-9
joining views, 2-44
joins, 1-42
OR operators, 2-29, 8-18
overview of, 1-5
plan stability, 7-2
preserving with plan stability, 7-2
TKPROF, 10-7, 10-10
viewing, 1-5
EXPLAIN PLAN statement
access paths, 1-28, 8-3, 8-4, 8-5, 8-6, 8-7, 8-8, 8-9, 8-10, 8-11, 8-12, 8-13, 8-14, 8-15
and domain indexes, 9-19
and full partition-wise joins, 9-17
and partial partition-wise joins, 9-16
and partitioned objects, 9-11
examples of output, 8-20, 10-16
invoking with the TKPROF program, 10-10
PLAN_TABLE table, 9-4
restrictions, 9-20
Export utility
copying statistics, 3-2
exporting data, 21-19
extensible optimization, 1-72
user-defined costs, 1-73
user-defined selectivity, 1-73
user-defined statistics, 1-72

F

FACT hint, 5-21
fact tables
star joins, 1-65
star queries, 1-65
fast full index scans, 1-37
FAST_START_IO_TARGET initialization parameter, 17-4, 17-8
FAST_START_MTTR_TARGET initialization parameter, 17-4, 17-8, 17-12
FAST_START_PARALLEL_ROLLBACK initialization parameter, 17-16
fast-start checkpoints
FAST_START_MTTR_TARGET initialization parameter, 17-5
LOG_CHECKPOINT_INTERVAL initialization parameter, 17-6
LOG_CHECKPOINT_TIMEOUT initialization parameter, 17-6
fast-start on-demand rollback, 17-15
fast-start parallel rollback, 17-15
features, new, xxxi
Fetch event, 12-15
FIRST_ROWS hint, 1-12
FIRST_ROWS(n) hint, 5-7
FORMAT statement
in Oracle Trace, 12-3
formatter tables
in Oracle Trace, 12-3
free buffer wait events, 22-34
free lists, 13-7
FULL hint, 4-6, 5-10
full partition-wise joins, 9-17
full table scans, 8-14, 8-21, 22-29
rule-based optimizer, 8-14
function-based indexes, 4-9

functions
PL/SQL
deterministic, 2-26
SQL
optimizing view queries, 2-41
user-defined
extensible optimization, 1-72

G

GATHER_ INDEX_STATS procedure
in DBMS_STATS package, 3-6
GATHER_DATABASE_STATS procedure
in DBMS_STATS package, 3-6
GATHER_SCHEMA_STATS procedure
in DBMS_STATS package, 3-6
GATHER_TABLE_STATS procedure
in DBMS_STATS package, 3-6
GETMISSES column
in V$ROWCACHE table, 14-32
GETS column
in V$ROWCACHE view, 14-32
global hints, 5-39
GLOGIN.SQL, 11-3, 11-7
GROUP BY clause
NOSORT clause, 14-59
optimizing views, 2-34

H

hash
distribution value, 9-24
hash clusters
scans of, 1-39, 8-4, 8-6
HASH hint, 5-11
hash joins
index join, 1-38
hash partitions, 9-11
examples of, 9-11
HASH_AJ hint, 1-64, 5-26, 5-27
HASH_AREA_SIZE initialization parameter, 1-69
HASH_JOIN_ENABLED initialization
parameter, 1-69
HASH_SJ hint, 1-65, 5-27
hashing, 4-21
HIGH_VALUE statistics, 1-40
hint
NL_AJ hint, 5-26
hints, 5-2
access paths, 5-9, 5-17
ALL_ROWS hint, 5-6
AND_EQUAL hint, 4-6, 5-17
as used in outlines, 7-3
CACHE hint, 5-34
cannot override sample access path, 1-39
CHOOSE hint, 5-8
CLUSTER hint, 5-11
CURSOR_SHARING_EXACT hint, 5-37, 1
degree of parallelism, 5-28
extensible optimization, 1-72
FACT hint, 5-21
FIRST_ROWS hint, 5-7
FIRST_ROWS(n) hint, 5-7
FULL hint, 4-6, 5-10
global, 5-39
HASH hint, 5-11
HASH_AJ hint, 5-26
HASH_SJ hint, 5-27
how to use, 5-2
INDEX hint, 4-6, 5-12, 5-22
INDEX_ASC hint, 5-13
INDEX_DESC hint, 5-14, 5-15
INDEX_FFS, 1-37
INDEX_JOIN, 1-38
join operations, 5-23
LEADING hint, 5-26
MERGE_AJ and HASH_AJ, 1-64
MERGE_AJ hint, 5-26
MERGE_SJ and HASH_SJ, 1-65
MERGE_SJ hint, 5-27
NL_AJ hint, 5-26
NL_SJ hint, 5-27
NO_EXPAND hint, 5-18
NO_FACT hint, 5-21
NO_INDEX, 4-6
NO_INDEX hint, 5-16
NO_MERGE hint, 5-20
NO_PUSH_PRED hint, 5-36
NO_UNNEST hint, 5-35
NOCACHE hint, 5-34

NOPARALLEL hint, 5-29
NOREWRITE hint, 5-19
optimization approach and goal, 5-6
ORDERED hint, 1-64, 5-22
overriding optimizer choice, 1-39
overriding OPTIMIZER_MODE and OPTIMIZER_GOAL, 1-12
PARALLEL hint, 5-28
parallel query option, 5-28
PQ_DISTRIBUTE hint, 5-29
PUSH_PRED hint, 5-36
PUSH_SUBQ hint, 5-36
REWRITE hint, 5-19
ROWID hint, 5-11
STAR hint, 5-22
UNNEST hint, 5-35
USE_CONCAT hint, 5-18
USE_MERGE hint, 5-24
USE_NL hint, 5-23
histograms, 3-20
number of buckets, 3-22
HOLD_CURSOR clause, 14-25

I

ID column
PLAN_TABLE table, 9-21
idle wait events, 22-43
SQL*Net message from client, 22-21
Import utility
copying statistics, 3-2
IN operator, 2-21
merging views, 2-35
IN subquery, 2-34
INDEX hint, 4-6, 4-14, 5-12
index joins, 1-38
INDEX_ASC hint, 5-13
INDEX_COMBINE hint, 4-6, 4-14
INDEX_DESC hint, 5-14, 5-15
INDEX_FFS hint, 1-37, 1-38
INDEX_JOIN hint, 1-38
indexes
avoiding the use of, 4-6
bitmap, 4-11, 4-12, 4-16
choosing columns for, 4-3
cluster
scans of, 8-7
composite, 4-4
scans of, 8-8
creating, 13-8
domain, 4-18
domain indexes
extensible optimization, 1-72
user-defined statistics, 1-72
dropping, 4-2
enforcing uniqueness, 4-8
ensuring the use of, 4-6
example, 8-21
function-based, 4-9
index joins, 1-38
modifying values of, 4-4
non-unique, 4-8
optimization and, 2-28, 8-17
placement on disk, 15-14
rebuilding, 4-7
recreating, 4-6
scans of, 1-29
bounded range, 8-9
cluster key, 8-7
composite, 8-8
MAX or MIN, 8-12
ORDER BY, 8-13
restrictions, 8-14
single-column, 8-8
unbounded range, 8-10
selectivity of, 4-3
statement conversion and, 2-28, 8-17
statistics, gathering, 3-8
indexing data, 13-8
information views, 24-3
initial database creation, 13-2
initialization files, 13-2, 13-9
initialization parameters
CONTROL_FILES, 13-10
CPU_COUNT, 17-16
DB_BLOCK_BUFFERS, 13-10
DB_BLOCK_SIZE, 13-3, 13-10
DB_DOMAIN, 13-10
DB_FILE_MULTIBLOCK_READ_COUNT, 1-64
DB_NAME, 13-3, 13-10

FAST_START_PARALLEL_ROLLBACK, 17-16
in Oracle Trace, 12-7
INITRANS, 13-7
JAVA_POOL_SIZE, 13-10
JOB_QUEUE_PROCESSES, 21-8
LOG_ARCHIVE_XXX, 13-10
LOG_CHECKPOINT_INTERVAL, 17-6
LOG_CHECKPOINT_TIMEOUT, 17-6
MAX_DUMP_FILE_SIZE, 10-4
OPEN_CURSORS, 13-10
OPTIMIZER_FEATURES_ENABLE, 1-37, 1-38, 2-34
OPTIMIZER_MODE, 1-11, 5-6, 8-2
PARALLEL_MAX_SERVERS, 17-7
PROCESSES, 13-10
RECOVERY_PARALLELISM, 17-7
SESSION_CACHED_CURSORS, 14-38
SESSIONS, 13-10
SHARED_POOL_SIZE, 13-10
SORT_AREA_SIZE, 1-64, 13-9
SQL_TRACE, 10-6
TIMED_STATISTICS, 10-4, 21-7
USER_DUMP_DEST, 10-4
INIT.ORA file
ORACLE_TRACE_ENABLE parameter, 12-35
INITRANS initialization parameter, 13-7
IN-lists, 5-13, 5-18
input parameters, table, 21-18
INPUT_IO item, 12-16
INSERT statement
append, 5-33
installation scripts
SPCREATE, 21-5
instance configuration, 13-9
instance numbers, 21-3
instrumentation
of Oracle Server, 12-15
INTERSECT operator
example, 2-48
optimizing view queries, 2-34
intra transaction recovery, 17-16
I/O
and SQL statements, 22-26
balancing, 15-3
excessive I/O waits, 22-26
objects causing I/O waits, 22-27
items
cross-product, 12-17
standard resource utilization, 12-16
types of, 12-16

J

JAVA_POOL_SIZE initialization parameter, 13-10
JOB_QUEUE_PROCESSES initialization parameter, 21-8
joins
anti-joins, 1-64
cluster, 8-4
searches on, 8-6
convert to subqueries, 2-31
execution plans and, 1-42
index joins, 1-38
join order
execution plans, 1-5
selectivity of predicates, 1-73, 3-2, 3-20
nested loops
cost-based optimization, 1-63
optimization of, 8-15
outer
non-null values for nulls, 2-43
parallel, and PQ_DISTRIBUTE hint, 5-29
partition-wise
examples of full, 9-17
examples of partial, 9-16
full, 9-17
sample table scan not supported, 1-27
select-project-join views, 2-33
semi-joins, 1-64
sort-merge
cost-based optimization, 1-63
example, 8-12
star joins, 1-65
star queries, 1-65

K

KEEP cache, 14-12
keys
searches, 8-4

L

LARGE_POOL_SIZE initialization
parameter, 14-34
latch free wait events
actions, 22-36
latches
tuning, 24-11
LEADING hint, 5-26
library cache
memory allocation, 14-32
LIKE operator, 2-20
Lmode modes, 24-18
load balancing, 15-3
loading data, 13-8
locally managed tablespaces, 21-4
location of initial datafile database option, 13-4
lock types, 24-17
ST (space transaction) locks, 24-18
TM (DML) locks, 24-18
TX (row transaction) locks, 24-17
UL (user defined) locks, 24-18
locking rows, 13-7
log buffer tuning, 14-45
log file switch wait events, 22-41
log writer processes
tuning, 15-16
LOG_ARCHIVE_XXX initialization
parameter, 13-10
LOG_BUFFER initialization parameter, 14-45
setting, 14-46
LOG_CHECKPOINT_INTERVAL initialization
parameter, 17-3
recovery time, 17-6
LOG_CHECKPOINT_TIMEOUT initialization
parameter, 17-4
recovery time, 17-6
LogicalTX event, 12-15
lookup tables
star queries, 1-65
LOW_VALUE statistics, 1-40
LRU
aging policy, 14-12
latch contention, 22-40

M

manual database creation, 13-2
max session memory statistic, 14-35
MAX_DISPATCHERS initialization
parameter, 19-4
MAX_DUMP_FILE_SIZE initialization
parameter, 10-4
SQL Trace, 10-4
MAX_SHARED_SERVERS initialization
parameter, 19-7
MAXOPENCURSORS clause, 14-25
MAXRS_SIZE item, 12-16
memory
reducing usage, 14-60
memory allocation
importance, 14-2
library cache, 14-32
shared SQL areas, 14-32
sort areas, 14-57
tuning, 14-4
MERGE hint, 5-19
MERGE_AJ hint, 1-64, 5-26, 5-27
MERGE_SJ hint, 1-65, 5-27
merging complex views, 2-34
merging views into statements, 2-33
migrated rows, 22-18
Migration event, 12-15
MINUS operator
optimizing view queries, 2-34
mirroring
redo logs, 15-17
modes
Lmode, 24-18
request, 24-18
multiple buffer pools, 14-11

N

NAMESPACE column
V$LIBRARYCACHE view, 14-27
national character set database option, 13-4
NCHAR datatype, 13-4
nested loops joins
cost-based optimization, 1-63

network
array interface, 23-13
detecting performance problems, 23-6
problem solving, 23-8
Session Data Unit, 23-14
tuning, 23-1
network bottlenecks, 22-21
network communication wait events, 22-20
DB file sequential/scattered read wait events, 22-25, 22-27
SQL*Net message from Dblink, 22-22
new features, xxxi
NL_AJ hint, 5-26
NL_SJ hint, 5-27
NLS_SORT initialization parameter
ORDER BY access path, 8-13
NO_EXPAND hint, 5-18
NO_FACT hint, 5-21
NO_INDEX hint, 4-6, 5-16
NO_MERGE hint, 5-20
NO_PUSH_PRED hint, 5-36
NO_UNNEST hint, 5-35
NOAPPEND hint, 5-34
NOCACHE hint, 5-34
NOPARALLEL hint, 5-29
NOPARALLEL_INDEX hint, 5-32
NOREWRITE hint, 5-19
NOSORT clause, 14-59
NOT IN subquery, 1-64
NOT operator, 2-23
NT performance, 16-6
nulls
non-null values for, 2-43
NUM_DISTINCT column
USER_TAB_COLUMNS view, 1-40
NUM_ROWS column
USER_TABLES view, 1-40
NVARCHAR datatype, 13-4
NVARCHAR2 datatype, 13-4

O

OBJECT_INSTANCE column
PLAN_TABLE table, 9-21
OBJECT_NAME column
PLAN_TABLE table, 9-21
OBJECT_NODE column
PLAN_TABLE table, 9-21
OBJECT_OWNER column
PLAN_TABLE table, 9-21
OBJECT_TYPE column
PLAN_TABLE table, 9-21
OEM
See Oracle Enterprise Manager (OEM)
OPEN_CURSORS initialization parameter, 13-10
increasing cursors per session, 14-32
operating system
data cache, 16-2
monitoring disk I/O, 22-8
OPERATION column
PLAN_TABLE table, 9-20, 9-25
OPTIMAL parameter, 18-3
optimization
choosing the approach, 1-11
conversion of expressions and predicates, 2-2
cost-based, 1-15
choosing an access path, 1-39
examples of, 1-40
histograms, 3-20
remote databases and, 2-13
star queries, 1-65
user-defined costs, 1-73
described, 1-3
DISTINCT, 2-34
distributed SQL statements, 2-12
extensible optimizer, 1-72
GROUP BY views, 2-34
hints, 1-12, 1-37, 1-38
manual, 1-12
merging complex views, 2-34
merging views into statements, 2-33
non-null values for nulls, 2-43
operations performed, 1-4
rule-based, 8-2, 8-15
selectivity of predicates, 3-2
histograms, 3-20
user-defined, 1-73
select-project-join views, 2-33
semi-joins, 1-64
statistics, 3-2

user-defined, 1-72
transitivity and, 2-23
without merging, 2-43
optimizer, 1-3
plan stability, 7-2
OPTIMIZER column
PLAN_TABLE, 9-21
OPTIMIZER_FEATURES_ENABLE initialization parameter, 1-37, 1-38, 1-66, 2-34
OPTIMIZER_GOAL clause, 1-12
OPTIMIZER_GOAL initialization parameter, 1-70
OPTIMIZER_INDEX_CACHING initialization parameter, 1-69
OPTIMIZER_INDEX_COST_ADJ initialization parameter, 1-69
OPTIMIZER_MAX_PERMUTATIONS initialization parameter, 1-70
OPTIMIZER_MODE initialization parameter, 1-11, 1-12, 1-70, 5-6, 8-2
hints affecting, 1-12
OPTIONS column
PLAN_TABLE table, 9-21
Oracle Enterprise Manager
Oracle Expert, 4-3
Oracle Index Tuning Wizard, 4-3
SQL Analyze, 4-3
Oracle Forms, 10-5
control of parsing and private SQL areas, 14-26
Oracle managed files, 15-18
Oracle Net Configuration Assistant, 23-14
Oracle Real Application Clusters
and Statspack, 21-22
Oracle Trace, 12-1
accessing collected data, 12-3
binary files, 12-3
collection results, 12-12
collections, 12-8
command-line interface, 12-3
deleting files, 12-7
duration events, 12-2
events, 12-2
FORMAT statement, 12-3
formatter tables, 12-3
parameters, 12-7
point events, 12-2
reporting utility, 12-14
START statement, 12-3, 12-4
STOP statement, 12-3, 12-6
ORACLE_TRACE_COLLECTION_NAME initialization parameter, 12-8
ORACLE_TRACE_COLLECTION_PATH initialization parameter, 12-8
ORACLE_TRACE_COLLECTION_SIZE initialization parameter, 12-8
ORACLE_TRACE_ENABLE initialization parameter, 12-8, 12-35
ORACLE_TRACE_FACILITY_NAME initialization parameter, 12-8, 12-9
ORACLE_TRACE_FACILITY_PATH initialization parameter, 12-8
Oracle-managed files
tuning, 15-18
ORDERED hint, 1-64, 5-22
ORDERED_PREDICATES hint, 5-37
OTHER column
PLAN_TABLE table, 9-22
OTHER_TAG column
PLAN_TABLE table, 9-21
outer joins
non-null values for nulls, 2-43
outlines
CREATE OUTLINE statement, 7-5
creating and using, 7-4
execution plans and plan stability, 7-2
hints, 7-3
moving tables, 7-9
storage requirements, 7-4
using, 7-6
using to move to the cost-based optimizer, 7-11
viewing data for, 7-9
OUTPUT_IO item, 12-16
overloaded disks, 15-9

P

page table, 16-10
PAGEFAULT_IO item, 12-16
PAGEFAULTS item, 12-16
paging, 16-10
reducing, 14-3

PARALLEL clause
  CREATE INDEX statement, 13-8
  RECOVER statement, 17-7
parallel execution
  hints, 5-28
PARALLEL hint, 5-28
parallel joins
  and PQ_DISTRIBUTE hint, 5-29
parallel recovery, 17-7
PARALLEL_BROADCAST_ENABLED initialization parameter, 1-71
PARALLEL_MAX_SERVERS initialization parameter, 17-7
parameter files, 13-2
PARENT_ID column
  PLAN_TABLE table, 9-21
Parse event, 12-15
parsing
  Oracle Forms, 14-26
  Oracle precompilers, 14-25
  reducing unnecessary calls, 14-24
PARTITION_ID column
  PLAN_TABLE table, 9-22
PARTITION_START column
  PLAN_TABLE table, 9-22
PARTITION_STOP column
  PLAN_TABLE table, 9-22
PARTITION_VIEW_ENABLED initialization parameter, 1-71
partitioned objects
  and EXPLAIN PLAN statement, 9-11
partitioning
  distribution value, 9-24
  examples of, 9-11
  examples of composite, 9-13
  hash, 9-11
  range, 9-11
  start and stop columns, 9-12
partitions
  statistics, 3-4
partition-wise joins
  full, 9-17
  full, and EXPLAIN PLAN output, 9-17
  partial, and EXPLAIN PLAN output, 9-16
PCTFREE parameter, 13-7, 22-18
PCTINCREASE parameter, 18-4
PCTUSED parameter, 13-7, 22-18
performance
  mainframe, 16-6
  NT, 16-6
  of SQL statements, 11-1
  reports
    generating, 21-8
    running, 21-3, 21-8
  UNIX-based systems, 16-5
  viewing execution plans, 1-5
Performance Monitor
  NT, 16-10
PERFSTAT user, 21-3, 21-4, 21-13
physical reads statistic, 14-8
PhysicalTX event, 12-15
plan stability, 7-2
  limitations of, 7-2
  preserving execution plans, 7-2
  procedures for the cost-based optimizer, 7-11
  use of hints, 7-2
PLAN_TABLE table, 11-2
  BYTES column, 9-21
  CARDINALITY column, 9-21
  COST column, 9-21
  DISTRIBUTION column, 9-22
  ID column, 9-21
  OBJECT_INSTANCE column, 9-21
  OBJECT_NAME column, 9-21
  OBJECT_NODE column, 9-21
  OBJECT_OWNER column, 9-21
  OBJECT_TYPE column, 9-21
  OPERATION column, 9-20
  OPTIMIZER column, 9-21
  OPTIONS column, 9-21
  OTHER column, 9-22
  OTHER_TAG column, 9-21
  PARENT_ID column, 9-21
  PARTITION_ID column, 9-22
  PARTITION_START column, 9-22
  PARTITION_STOP column, 9-22
  POSITION column, 9-21
  REMARKS column, 9-20
  SEARCH_COLUMNS column, 9-21
  STATEMENT_ID column, 9-20

TIMESTAMP column, 9-20
PL/SQL
deterministic functions, 2-26
PLUSTRACE role, 11-2
point events
in Oracle Trace, 12-2, 12-15
POOL attribute, 19-4
POSITION column
PLAN_TABLE table, 9-21
PQ_DISTRIBUTE hint, 5-29
precompilers
control of parsing and private SQL areas, 14-25
predicates
pushing into a view, 2-36, 2-41
examples, 2-36, 2-39
selectivity, 3-2
histograms, 3-20
user-defined, 1-73
PRIMARY KEY constraint, 4-8
primary keys
optimization, 2-32
searches, 8-5
PRIVATE_SGA variable, 14-36
procedures
DBMS_JOB, 21-7
DBMS_JOB.INTERVAL, 21-8
deterministic functions, 2-26
STATSPACK.MODIFY_STATSPACK_PARAMETER, 21-15, 21-17
STATSPACK.SNAP, 21-6, 21-17
process
dispatcher process configuration, 19-4
processes
priority, 16-3
scheduler, 16-3
scheduling, 16-11
PROCESSES initialization parameter, 13-10
program global area (PGA)
direct path read, 22-28
direct path write, 22-30
shared servers, 14-35, 19-2
pseudocolumns
ROWNUM
cannot use indexes, 8-15
optimizing view queries, 2-34, 2-43
PUSH_PRED hint, 5-36

Q

queries
avoiding the use of indexes, 4-6
compound
optimization of, 2-46
ORs converted to, 2-28, 8-17
ensuring the use of indexes, 4-6
optimizing IN subquery, 2-34
SAMPLE clause
cost-based optimization, 1-14
star queries, 1-65
QUERY_REWRITE_ENABLED initialization parameter, 1-71

R

range
distribution value, 9-24
range partitions, 9-11
examples of, 9-11
raw device, 16-2
read consistency, 22-17
read events
direct path, 22-28
read wait events
scattered, 22-25
REBUILD clause, 4-7
RECOVER statement
PARALLEL clause, 17-7
recovery
parallel
intra transaction recovery, 17-16
parallel processes for, 17-7
PARALLEL_MAX_SERVERS initialization parameter, 17-7
setting number of processes to use, 17-7
RECOVERY_PARALLELISM initialization parameter, 17-7
recursive calls, 10-14
RECYCLE cache, 14-12
REDO BUFFER ALLOCATION RETRIES statistic, 14-45

redo logs, 13-5
  mirroring, 15-17
  placement on disk, 15-15
  sizing, 13-5
reducing
  contention
    dispatchers, 19-3
    OS processes, 16-3
    shared servers, 19-5
  data dictionary cache misses, 14-32
  paging and swapping, 14-3
  rollback segment contention, 18-2
  unnecessary parse calls, 14-24
RELEASE_CURSOR clause, 14-25
REMARKS column
  PLAN_TABLE table, 9-20
removing data, 21-20
removing snapshots, 21-20
REPORT_NAME variable, 21-11
reports
  performance, 21-3, 21-8
  Statspack, 21-9
request modes, 24-18
resource bottlenecks, 22-22
resource wait events, 22-27
response time, 1-10
  cost-based approach, 1-11
  optimizing, 1-10, 5-7
REWRITE hint, 5-18
rollback segments, 22-17
  assigning to transactions, 18-3
  choosing how many, 18-2
  creating, 18-2
rollback tablespaces, 13-6
rollbacks
  fast-start on-demand, 17-15
  fast-start parallel, 17-15
round-robin
  distribution value, 9-24
row locking, 13-7
row sampling, 3-4
row sources, 1-8
row transaction locks, 24-17
ROWID hint, 5-11
rowids
  mapping to bitmaps, 4-15
  table access by, 1-28
ROWNUM pseudocolumn
  cannot use indexes, 8-15
  optimizing view queries, 2-34, 2-43
rows
  row sources, 1-8
  rowids used to locate, 1-28, 8-3
RowSource event, 12-2, 12-15
RULE hint
  OPTIMIZER_MODE and, 1-12
rule-based optimization, 8-2

S

SAMPLE BLOCK clause, 1-27
  access path, 1-28
    hints cannot override, 1-39
SAMPLE clause, 1-27
  access path, 1-28
    hints cannot override, 1-39
  cost-based optimization, 1-14
sample table scans, 1-27, 1-28
  hints cannot override, 1-39
sar UNIX command, 16-10
scans
  cluster, 8-4, 8-6
    indexed, 8-7
  full table, 8-14
    rule-based optimizer, 8-14
  hash cluster, 8-4, 8-6
  index, 1-29
    bitmap, 1-38
    bounded range, 8-9
    cluster key, 8-7
    composite, 8-8
    MAX or MIN, 8-12
    ORDER BY, 8-13
    restrictions, 8-14
    single-column, 8-8
    unbounded range, 8-10
  index joins, 1-38
  range, 8-8
    bounded, 8-9
    MAX or MIN, 8-12

ORDER BY, 8-13
unbounded, 8-10
sample table, 1-27, 1-28
hints cannot override, 1-39
unique, 8-5, 8-7
scattered read wait events, 22-25
actions, 22-25
schemas
star schemas, 1-65
SCPU item, 12-16
scripts
SPAUTO.SQL, 21-8
SPCPKG, 21-5
SPCREATE, 21-5
SPCTAB, 21-5
SPCUSR, 21-5
SPPURGE.SQL, 21-20
SPTRUNC.SQL, 21-22
Statspack documentation scripts, 21-25
Statspack installation scripts, 21-24
Statspack performance data maintenance scripts, 21-25
Statspack reporting and automation scripts, 21-24
Statspack supplied scripts, 21-24
upgrading Statspack scripts, 21-24
SEARCH_COLUMNS column
PLAN_TABLE table, 9-21
SELECT statement
SAMPLE clause, 1-27
access path, 1-28, 1-39
cost-based optimization, 1-14
selectivity, 3-2
histograms, 3-20
index, 4-3
user-defined, 1-73
select-project-join views, 2-33
semi-joins, 1-64
Session Data Unit (SDU), 23-14
session id, 21-17
session memory statistic, 14-35
SESSION_CACHED_CURSORS initialization parameter, 14-38
SESSIONS initialization parameter, 13-10
SET AUTOTRACE, 11-2
SET TRANSACTION statement, 18-3
SGA size, 14-45
shared server
performance issues, 19-2
reducing contention, 19-2
tuning, 19-2
tuning memory, 14-34
shared SQL areas
memory allocation, 14-32
SHARED_POOL_RESERVED_SIZE initialization parameter, 14-40
SHARED_POOL_SIZE initialization parameter, 13-10, 14-32, 14-40
allocating library cache, 14-32
tuning the shared pool, 14-36
sharing data, 21-19
SHOW SGA statement, 14-4
sizing redo logs, 13-5
snapshot levels, 21-14, 21-16
snapshot thresholds, 21-14, 21-16
snapshots, 21-2
begin, end, 21-9
database identifier (DBID), 21-3
deleting, 21-20
instance numbers, 21-3
levels, 21-14, 21-16
removing, 21-20
SNAP_ID, 21-3
taking snapshots, 21-6
thresholds, 21-14, 21-16
SOME operator, 2-21
sort areas
memory allocation, 14-57
sort merge joins
access path, 8-12
cost-based optimization, 1-63
example, 8-12
SORT_AREA_SIZE initialization parameter, 1-71, 4-13, 13-9
cost-based optimization and, 1-64
tuning sorts, 14-58
sorts
(disk) statistic, 14-56
(memory) statistic, 14-56
avoiding on index creation, 14-59

space transaction locks, 24-18
SPAUTO.SQL script, 21-8
SPCPKG script, 21-5
SPCREATE script, 21-5
SPCTAB script, 21-5
SPCUSR script, 21-5
SPPURGE.SQL script, 21-20
SPREPORT.SQL file, 21-9
SPTRUNC.SQL script, 21-22
SQL
  functions
    optimizing view queries, 2-41
SQL Parse event, 12-2
SQL statements
  avoiding the use of indexes, 4-6
  complex, 2-31
    optimizing, 2-31
  converting
    examples of, 2-27, 8-17
  distributed
    optimization of, 2-12
  ensuring the use of indexes, 4-6
  execution plans of, 1-5
  modifying indexed data, 4-4
  optimization
    complex statements, 2-31
  thresholds, 21-14, 21-16
  waiting for I/O, 22-26
SQL trace facility, 10-2, 10-6
  example of output, 10-16
  output, 10-12
  statement truncation, 10-14
  steps to follow, 10-3
  trace files, 10-4
SQL*Loader, 13-8
SQL*Net message from client idle events, 22-21
SQL*Net message from dblink wait events, 22-22
SQL*Plus
  variables
    BEGIN_SNAP, 21-11
    DEFAULT_TABLESPACE, 21-6
    END_SNAP, 21-11
    REPORT_NAME, 21-11
    TEMPORARY_TABLESPACE, 21-6
SQL_STATEMENT column
    TKPROF_TABLE, 10-18
SQL_TRACE initialization parameter, 10-6
SQL.BSQ file, 13-4
SQLSegment event, 12-15
ST locks, 24-18
standard resource utilization items, 12-16
STAR hint, 5-22
star joins, 1-65
star query, 1-65
star transformation, 5-20
STAR_TRANSFORMATION hint, 5-20
STAR_TRANSFORMATION_ENABLED
    initialization parameter, 1-71, 5-21
start columns
  in partitioning and EXPLAIN PLAN
      statement, 9-12
START statement in Oracle Trace, 12-3, 12-4
STATEMENT_ID column
  PLAN_TABLE table, 9-20
statistics, 11-3
  automated collecting, 21-7
  automated gathering, 21-7
  collecting, 21-7
  collection interval, 21-8
  consistent gets, 14-8, 18-2
  db block gets, 14-8, 18-2
  estimated
    block sampling, 3-4
    row sampling, 3-4
  exporting and importing, 3-2
  extensible optimization, 1-72
  from ANALYZE, 3-12
  from B-tree or bitmap index, 3-8
  gathering with DBMS_STATS package, 3-5
  generating, 3-3
  generating and managing with DBMS_
      STATS, 3-5
  generating for cost-based optimization, 3-3
  HIGH_VALUE and LOW_VALUE, 1-40
  max session memory, 14-35
  optimizer mode, 1-11
  optimizer use of, 1-15, 3-2
  partitions and subpartitions, 3-4
  physical reads, 14-8
  selectivity of predicates, 3-2

histograms, 3-20
user-defined, 1-73
session memory, 14-35
shared server processes, 19-5
sorts (disk), 14-56
sorts (memory), 14-56
user-defined statistics, 1-72
Statspack
and Oracle Real Application Clusters, 21-22
documentation scripts, 21-25
exporting data, 21-19
installation
batch mode, 21-6
interactive, 21-4
installation scripts, 21-24
performance data maintenance scripts, 21-25
removing, 21-23
reporting and automation scripts, 21-24
running reports, 21-9
scripts, 21-24
sharing data, 21-19
space requirements, 21-3
uninstalling, 21-23
upgrading scripts, 21-24
vs. BSTAT/ESTAT, 20-5, 21-3
STATSPACK.MODIFY_STATSPACK_PARAMETER procedure, 21-15, 21-17
STATSPACK.SNAP procedure, 21-6, 21-17
stop columns
in partitioning and EXPLAIN PLAN statement, 9-12
STOP statement in Oracle Trace, 12-3, 12-6
STORAGE clause
OPTIMAL parameter, 18-3
stored outlines
creating and using, 7-4
execution plans and plan stability, 7-2
hints, 7-3
moving tables, 7-9
storage requirements, 7-4
using, 7-6
viewing data for, 7-9
striping, 15-3
manual, 15-14
subpartitions
statistics, 3-4
subqueries
converting to joins, 2-31
NOT IN, 1-64
optimizing IN subquery, 2-34
subquery unnesting, 6-19
swapping, 16-10
reducing, 14-3
switching processes, 16-11
System Global Area tuning, 14-4
system statistics, gathering, 3-6

T

tables
creating, 13-7
dimensions
star queries, 1-65
fact tables
star queries, 1-65
formatter in Oracle Trace, 12-3
free list settings, 13-7
full scans, 22-29
lookup tables, 1-65
placement on disk, 15-14
setting storage options, 13-7
tablespaces, 13-6
creating, 13-6
dictionary managed, 21-4
locally managed, 21-4
rollback, 13-6
temporary, 13-6
TCP.NODELAY parameter, 23-14
temporary tablespaces, 13-6
TEMPORARY_TABLESPACE variable, 21-6
thrashing, 16-10
thread, 16-3
thresholds, SQL statement, 21-14, 21-16
throughput, 1-10
cost-based approach, 1-11
optimizing, 1-10, 5-6
TIMED_STATISTICS initialization parameter, 10-4, 16-2, 21-7
SQL Trace, 10-4
TIMESTAMP column

PLAN_TABLE table, 9-20
TKPROF program, 10-3, 10-6
editing the output SQL script, 10-17
example of output, 10-16
generating the output SQL script, 10-16
syntax, 10-8
using the EXPLAIN PLAN statement, 10-10
TKPROF_TABLE, 10-18
querying, 10-17
TM locks, 24-18
Trace, Oracle, 12-1
tracing statements
for performance statistics, 11-3
for query execution path, 11-3
using a database link, 11-6
with parallel query option, 11-7
transactions
assigning rollback segments, 18-3
truncating data, 21-22
tuning, 11-1
latches, 24-11
logical structure, 4-2
memory allocation, 14-4
resource contention, 22-1
shared server, 19-2
System Global Area (SGA), 14-4
TX locks, 24-17

U

UCPU item, 12-16
UL locks, 24-18
UNION ALL operator
examples, 2-29, 2-31, 2-46, 8-18, 8-20
optimizing view queries, 2-34
transforming OR into, 2-28, 8-17
UNION operator
examples, 2-36, 2-47
optimizing view queries, 2-34
UNIQUE constraint, 4-8
unique keys
optimization, 2-32
searches, 8-5
uniqueness, 4-8
UNIX system performance, 16-5
UNNEST hint, 5-35
UNNEST_SUBQUERY parameter, 5-35
upgrade
to the cost-based optimizer, 7-12
USE_CONCAT hint, 5-18
USE_MERGE hint, 5-24
USE_NL hint, 5-23
USE_STORED_OUTLINES parameter, 7-6
user defined locks, 24-18
user global area (UGA)
shared servers, 14-34, 19-2
V$SESSTAT, 14-35
USER_DUMP_DEST initialization parameter, 10-4
SQL Trace, 10-4
USER_ID column
TKPROF_TABLE, 10-18
USER_OUTLINE_HINTS view
stored outline hints, 7-9
USER_OUTLINES view
stored outlines, 7-9
USER_TAB_COL_STATISTICS view, 1-41
USER_TAB_COLUMNS view, 1-40, 1-41
USER_TABLES view, 1-40
user-defined costs, 1-73
UTLCHN1.SQL script, 22-18

V

V$BH view, 14-13
V$BUFFER_POOL_STATISTICS view, 14-13
V$DATAFILE view, 24-7
V$DB_CACHE_ADVICE view, 14-5, 14-8, 14-9, 14-10, 14-11, 14-13
V$DB_OBJECT_CACHE view, 24-4
V$FAST_START_SERVERS view, 17-16
V$FAST_START_TRANSACTIONS view, 17-16
V$FILESTAT view, 24-6
V$INSTANCE_RECOVERY view, 17-8
V$LATCH view, 24-9
V$LATCH_CHILDREN view, 24-12
V$LATCH_HOLDER view, 24-13
V$LIBRARYCACHE view, 24-15
NAMESPACE column, 14-27
V$LOCK view, 24-17
V$MYSTAT view, 24-22

V$OPEN_CURSOR view, 24-23
V$PARAMETER view, 24-25
V$PROCESS view, 24-27
V$QUEUE view, 19-5
V$ROLLSTAT view, 24-29
V$ROWCACHE view, 24-46
  GETMISSES column, 14-32
  GETS column, 14-32
  performance statistics, 14-30
V$RSRC_CONSUMER_GROUP view, 22-7
V$SESSION view, 24-32
V$SESSION_EVENT view, 24-36
  network information, 23-6
V$SESSION_WAIT view, 22-10, 24-38
  network information, 23-6
V$SESSTAT view, 22-7, 24-46
  network information, 23-6
  statistics, 24-42
  using, 14-35
V$SHARED_POOL_RESERVED view, 14-40
V$SQL view, 24-46
V$SQL_PLAN view, 24-47
V$SQLAREA view, 24-53
V$SQLTEXT view, 24-56
V$SYSSTAT view, 24-46
  redo buffer allocation, 14-45
  statistics, 24-58
  tuning sorts, 14-56
  using, 14-8
V$SYSTEM_EVENT view, 24-63
V$SYSTEM_PARAMETER view, 24-25
V$UNDOSTAT view, 13-11, 24-66
V$WAITSTAT view, 22-10, 24-68
VARCHAR datatype, 13-4
VARCHAR2 datatype, 13-4
variables
  bind variables
    optimization, 1-41
views
  complex view merging, 2-34
  counter/accumulator, 24-2
  current state views, 24-2
  histograms, 3-24
  information views, 24-3
  non-null values for nulls, 2-43
  select-project-join views, 2-33
  statistics, 3-15
vmstat UNIX command, 16-10

W

wait events
  buffer busy waits, 22-22
  contention wait events, 22-36
  direct path, 22-30
  event timings, 21-19
  free buffer waits, 22-34
  idle wait events, 22-43
  log file switch, 22-41
  network communication wait events, 22-20
  reasons for, 24-68
  resource wait events, 22-27
  time units, 21-18
wait time description, 24-38