Oracle® Database

Performance Tuning Guide

11*g* Release 1 (11.1)

B28274-01

July 2007

ORACLE®

Oracle Database Performance Tuning Guide, 11*g* Release 1 (11.1)

B28274-01

Primary Author: Immanuel Chan

Contributors: Aditya Agrawal, Lance Ashdown, Hermann Baer, James Barlow, Vladimir Barriere, Mehul Bastawala, Ruth Baylis, Eric Belden, Pete Belknap, Supiti Buranawatanachoke, Qiang Cao, Sunil Chakkappen, Sumanta Chatterjee, Maria Colgan, Alvaro Corena, Benoit Dageville, Dinesh Das, Karl Dias, Vinayagam Djegaradjane, Harvey Eneman, Bjorn Engsig, Mike Feng, Leonidas Galanis, Cecilia Gervasio, Bhaskar Ghosh, Ray Glasstone, Leslie Gloyd, Prabhaker Gongloor, Kiran Goyal, Connie Dialeris Green, Russell Green, Joan Gregoire, Lester Gutierrez, Lex de Haan, Karl Haas, Brian Hirano, Lilian Hobbs, William Hodak, Andrew Holdsworth, Mamdouh Ibrahim, Hakan Jacobsson, Christopher Jones, Srinivas Kareenhalli, Feroz Khan, Stella Kister, Sergey Koltakov, Vivekanada Kolla, Paul Lane, Sue K. Lee, Herve Lejeune, Yunrui Li, Juan Loaiza, Diana Lorentz, George Lumpkin, Joe McDonald, Bill McKenna, Mughees Minhas, Valarie Moore, Sujatha Muthulingam, Gary Ngai, Michael Orlowski, Kant C. Patel, Richard Powell, Mark Ramacher, Shankar Raman, Yair Sarig, Uri Shaft, Vishwanath Sreeraman, Vinay Srihari, Sankar Subramanian, Margaret Susairaj, Hal Takahara, Nick Taylor, Misha Tyulenev, Amir Valiani, Mark Van de Wiel, Venkateshwaran Venkataramani, Nitin Vengurlekar, Stephen Vivian, Yujun Wang, Simon Watt, Andrew Witkowski, Keith Wong, Graham Wood, Khaled Yagoub, Mohamed Zait

Contents

3 Performance Improvement Methods

6 Automatic Performance Diagnostics

7 Memory Configuration and Use

8 I/O Configuration and Design

9 Understanding Operating System Resources

10 Instance Tuning Using Performance Views

Part IV Optimizing SQL Statements

11 The Query Optimizer

12 Using EXPLAIN PLAN

13 Managing Optimizer Statistics

14 Using Indexes and Clusters

15 Using SQL Plan Management

16 SQL Tuning Overview

17 Automatic SQL Tuning

18 SQL Access Advisor

19 Using Optimizer Hints

20 Using Plan Stability

21 Using Application Tracing Tools

Preface

This preface contains these topics:

Audience

Oracle Database Performance Tuning Guide is an aid for people responsible for the operation, maintenance, and performance of Oracle. 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 book could be useful for database administrators, application designers, and programmers.

For information about using Oracle Enterprise Manager to tune the performance of Oracle Database, see *Oracle Database 2 Day + Performance Tuning Guide*.

Documentation Accessibility

Our goal is to make Oracle products, services, and supporting documentation accessible, with good usability, to the disabled community. 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. Accessibility 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 more information, visit the Oracle Accessibility Program Web site at

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

Accessibility of Code Examples in Documentation

Screen readers 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, some screen readers may not always read a line of text that consists solely of a bracket or brace.

Accessibility of Links to External Web Sites in Documentation

This documentation may contain links to Web sites of other companies or organizations that Oracle does not own or control. Oracle neither evaluates nor makes any representations regarding the accessibility of these Web sites.

TTY Access to Oracle Support Services

Oracle provides dedicated Text Telephone (TTY) access to Oracle Support Services within the United States of America 24 hours a day, seven days a week. For TTY support, call 800.446.2398.

Related Documents

Before reading this manual, you should have already read *Oracle Database Concepts*, *Oracle Database 2 Day DBA*, *Oracle Database Application Developer's Guide - Fundamentals*, and the *Oracle Database Administrator's Guide.*

For information about using Oracle Enterprise Manager to tune the performance of Oracle Database, see *Oracle Database 2 Day + Performance Tuning Guide*.

For more information about tuning data warehouse environments, see *Oracle Database Data Warehousing Guide*.

Many of the examples in this book use the sample schemas, which are installed by default when you select the Basic Installation option with an Oracle Database installation. Refer to *Oracle Database Sample Schemas* for information on how these schemas were created and how you can use them yourself.

For information about Oracle Database error messages, see *Oracle Database Error Messages*. Oracle Database error message documentation is only available in HTML. If you are accessing the error message documentation on the Oracle Documentation CD, you can browse the error messages by range. After you find the specific range, use your browser's find feature to locate the specific message. When connected to the Internet, you can search for a specific error message using the error message search feature of the Oracle online documentation.

Conventions

The following text conventions are used in this document:

Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.
italic	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
`monospace`	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.

What's New in Oracle Performance?

This section describes new performance tuning features of Oracle Database 11*g* Release 1 (11.1) and provides pointers to additional information. The features and enhancements described in this section comprise the overall effort to optimize database performance.

For a summary of all new features for Oracle Database 11*g* Release 1 (11.1), see *Oracle Database New Features Guide*.

The new and updated performance tuning features in Oracle Database 11*g* Release 1 (11.1) include:

- Active session history (ASH) enhancements

  ASH statistics are enhanced to provide row-level activity information for each SQL statement that is captured. This information can be used to identify which part of the SQL execution contributed most significantly to the SQL elapsed time.

  For more information, see Chapter 5, "Automatic Performance Statistics".

- Automatic Database Diagnostic Monitor (ADDM) enhancements

  ADDM is enhanced to perform analysis:

  – On database clusters (Global ADDM)

  – On various levels of granularity, such as database cluster, database instance, or specific targets

  – Over any specified time period (not tied to a pair of snapshots)

  For more information, see Chapter 6, "Automatic Performance Diagnostics".

- Automated SQL tuning

  You can automate the scheduling of SQL Tuning Advisor tasks to run during maintenance windows using Automated Maintenance Task. For more information, see Chapter 17, "Automatic SQL Tuning".

- AWR baselines

  A baseline contains performance data from a specific time period that is preserved for comparison with other similar workload periods when performance problems occur. There are several types of available baselines in Oracle Database: fixed baselines, moving window baseline, and baseline templates. For more information, see Chapter 5, "Automatic Performance Statistics".

- Database Replay

You can capture a database workload on a production system and replay it on a test system to ensure that system changes, such as database upgrades, will yield desired results. For more information, see Chapter 22, "Database Replay".

- Enhanced I/O statistics

  I/O statistics are collected for all I/O calls made by Oracle Database in the following dimensions: consumer group, database file, and database function. For more information, see Chapter 10, "Instance Tuning Using Performance Views".

- Enhanced optimizer statistics maintenance

  The collection and publication of statistics is now decoupled. You can collect optimizer statistics, store them as pending statistics, test them, and then publish them only if the statistics are valid. For more information, see Chapter 13, "Managing Optimizer Statistics".

- Extended statistics

  The optimizer now gathers information on a group of columns in a table (MultiColumn statistics) to analyze column dependencies in column groups. It also gathers information about expressions in columns. For more information, see Chapter 13, "Managing Optimizer Statistics".

- I/O calibration

  The I/O calibration feature of Oracle Database enables you to assess the performance of the storage subsystem, and determine whether I/O performance problems are caused by the database or the storage subsystem. For more information, see Chapter 8, "I/O Configuration and Design".

- Query result caches

  You can store results of frequently run SQL queries in server-side and client-side result caches. This ensures a faster response time when these queries are subsequently executed. For more information, see Chapter 7, "Memory Configuration and Use".

- Real-time SQL monitoring

  Real-time SQL monitoring enables you to monitor the execution of long running SQL statements as they are being executed. Both cursor-level statistics (such as CPU time and I/O time) and execution plan statistics (such as cardinality of intermediate results, memory and temporary space used by each operator in the plan) are updated in near real-time while the statement executes. These statistics are exposed by two new fixed views, `V$SQL_MONITOR` and `V$SQL_PLAN_MONITOR`. For more information, see Chapter 10, "Instance Tuning Using Performance Views".

- SQL Performance Analyzer

  The SQL Performance Analyzer enables you to forecast the impact of system changes on SQL performance by testing these changes using a SQL workload on a test system. For more information, see Chapter 23, "SQL Performance Analyzer".

- SQL plan baselines

  Changes to the execution plan for a SQL statement can cause severe performance degradation. Oracle Database can capture, select, and evolve SQL plan baselines, preventing the optimizer from using a new plan until it has been verified to be more efficient than the existing plan. This feature replaces the plan stability feature in previous versions of Oracle Database. For more information, see Chapter 15, "Using SQL Plan Management".

- SQL test case builder

  The SQL test case builder provides the ability to build a reproducible test case by gathering and packaging the necessary information related to an SQL incident so that it can be reproduced on another system. For more information, see Chapter 16, "SQL Tuning Overview".

Part I

Performance Tuning

Part I provides an introduction and overview of performance tuning.

The chapter in this part is:

- Chapter 1, "Performance Tuning Overview"

1

Performance Tuning Overview

This chapter provides an introduction to performance tuning and contains the following sections:

- Introduction to Performance Tuning
- Introduction to Performance Tuning Features and Tools

Introduction to Performance Tuning

This guide provides information on tuning an Oracle Database system for performance. Topics discussed in this guide include:

- Performance Planning
- Instance Tuning
- SQL Tuning

> **See Also:** *Oracle Database 2 Day + Performance Tuning Guide* for information about using Oracle Enterprise Manager to tune the performance of Oracle Database

Performance Planning

Before starting on the instance or SQL tuning sections of this guide, make sure you have read Part II, "Performance Planning". Based on years of designing and performance experience, Oracle has designed a performance methodology. This brief section explains clear and simple activities that can dramatically improve system performance. It discusses the following topics:

- Understanding Investment Options
- Understanding Scalability
- System Architecture
- Application Design Principles
- Workload Testing, Modeling, and Implementation
- Deploying New Applications

Instance Tuning

Part III, "Optimizing Instance Performance" of this guide discusses the factors involved in the tuning and optimizing of an Oracle database instance.

When considering instance tuning, care must be taken in the initial design of the database system to avoid bottlenecks that could lead to performance problems. In addition, you need to consider:

- Allocating memory to database structures
- Determining I/O requirements of different parts of the database
- Tuning the operating system for optimal performance of the database

After the database instance has been installed and configured, you need to monitor the database as it is running to check for performance-related problems.

Performance 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 database administrators (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 peak periods at the site and install a monitoring tool that gathers performance data for those high-load 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.

Ideally, baseline data gathered should include the following:

- Application statistics (transaction volumes, response time)
- Database statistics
- Operating system statistics
- Disk I/O statistics
- Network statistics

In the Automatic Workload Repository, baselines are identified by a range of snapshots that are preserved for future comparisons. See "Overview of the Automatic Workload Repository" on page 5-8.

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.

When to Tune

There are two distinct types of tuning:

- Proactive Monitoring
- 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 can also be considered as 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.

Experimenting with or 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 changes in the way the application is being used, and the way the application is using the database and host resources.

Bottleneck Elimination Tuning usually implies fixing a performance problem. However, tuning should be part of the life cycle of an application—through the analysis, design, coding, production, and maintenance stages. Oftentimes, 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 in the sections that follow.

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.

SQL Tuning

Part IV, "Optimizing SQL Statements" of this guide discusses the process of tuning and optimizing SQL statements.

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 are required, then Oracle can construct an accurate plan to access those rows efficiently through the shortest possible path. In decision support system (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 *Oracle Database Data Warehousing Guide*.

Query Optimizer and Execution Plans

When a SQL statement is executed on an Oracle database, the Oracle query optimizer determines the most efficient execution plan after considering many factors related to the objects referenced and the conditions specified in the query. This determination is an important step in the processing of any SQL statement and can greatly affect execution time.

During the evaluation process, the query optimizer reviews statistics gathered on the system to determine the best data access path and other considerations. You can override the execution plan of the query optimizer with hints inserted in SQL statement.

Introduction to Performance Tuning Features and Tools

Effective data collection and analysis is essential for identifying and correcting performance problems. Oracle provides a number of tools that allow a performance engineer to gather information regarding database performance. In addition to gathering data, Oracle provides tools to monitor performance, diagnose problems, and tune applications.

The Oracle gathering and monitoring features are mainly automatic, managed by an Oracle background processes. To enable automatic statistics collection and automatic performance features, the `STATISTICS_LEVEL` initialization parameter must be set to `TYPICAL` or `ALL`. You can administer and display the output of the gathering and tuning tools with Oracle Enterprise Manager, or with APIs and views. For ease of use and to take advantage of its numerous automated monitoring and diagnostic tools, Oracle Enterprise Manager Database Control is recommended.

See Also:

- *Oracle Database 2 Day DBA* for information on monitoring, diagnosing, and tuning the database
- *Oracle Enterprise Manager Concepts* for information about monitoring and diagnostic tools available with Oracle Enterprise Manager
- *Oracle Database PL/SQL Packages and Types Reference* for detailed information on the `DBMS_ADVISOR`, `DBMS_SQLTUNE`, and `DBMS_WORKLOAD_REPOSITORY` packages
- *Oracle Database Reference* for information on the `STATISTICS_LEVEL` initialization parameter

Automatic Performance Tuning Features

The Oracle automatic performance tuning features include:

- Automatic Workload Repository (AWR) collects, processes, and maintains performance statistics for problem detection and self-tuning purposes. See "Overview of the Automatic Workload Repository" on page 5-8.
- Automatic Database Diagnostic Monitor (ADDM) analyzes the information collected by the AWR for possible performance problems with the Oracle database. See "Overview of the Automatic Database Diagnostic Monitor" on page 6-1.
- SQL Tuning Advisor allows a quick and efficient technique for optimizing SQL statements without modifying any statements. See "Reactive Tuning Using the SQL Tuning Advisor" on page 17-9.
- SQLAccess Advisor provides advice on materialized views, indexes, and materialized view logs. See "Automatic SQL Tuning Features" on page 16-5 and "Overview of the SQL Access Advisor" on page 18-1 for information on SQLAccess Advisor.
- End to End Application tracing identifies excessive workloads on the system by specific user, service, or application component. See "End to End Application Tracing" on page 21-1.
- Server-generated alerts automatically provide notifications when impending problems are detected. See *Oracle Database Administrator's Guide* for information about monitoring the operation of the database with server-generated alerts.
- Additional advisors that can be launched from Oracle Enterprise Manager, such as memory advisors to optimize memory for an instance. The memory advisors are commonly used when automatic memory management is not set up for the database. Other advisors are used to optimize mean time to recovery (MTTR), shrinking of segments, and undo tablespace settings. See *Oracle Enterprise Manager Concepts* for information on advisors that are available with Oracle Enterprise Manager.

  To access the advisors through Oracle Enterprise Manager Database Control:

  - Click the **Advisor Central** link under **Related Links** at the bottom of the **Database** pages.
  - On the **Advisor Central** page, click one of the advisor links.
- Oracle Enterprise Manager Performance page displays host, instance service time, and throughput information for real time monitoring and diagnosis. The page can

be set to refresh automatically in selected intervals or manually. See *Oracle Enterprise Manager Concepts* for information on the Performance page available with Oracle Enterprise Manager.

Additional Oracle Tools

This section describes additional Oracle tools that can be used for determining performance problems.

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 the views that represent them, are automatically maintained by Oracle throughout the life of the instance. See Chapter 10, "Instance Tuning Using Performance Views" for information diagnosing tuning problems using the V$ performance views.

See Also: *Oracle Database Reference* for information about dynamic performance views

---

Note: Oracle recommends using the Automatic Workload Repository to gather performance data. These tools have been designed to capture all of the data needed for performance analysis.

Part II

Performance Planning

Part II describes ways to improve Oracle performance by starting with sound application design and using statistics to monitor application performance. It explains the Oracle Performance Improvement Method, as well as emergency performance techniques for dealing with performance problems.

The chapters in this part include:

- Chapter 2, "Designing and Developing for Performance"
- Chapter 3, "Performance Improvement Methods"

2

Designing and Developing for Performance

Optimal system performance begins with design and continues throughout the life of your system. Carefully consider performance issues during the initial design phase, and it will be easier to tune your system during production.

This chapter contains the following sections:

Oracle Methodology

System performance has become increasingly important as computer systems get larger and more complex as the Internet plays a bigger role in business applications. In order to accommodate this, Oracle has produced a performance methodology based on years of designing and performance experience. This methodology explains clear and simple activities that can dramatically improve system performance.

Performance strategies vary in their effectiveness, and systems with different purposes—such as operational systems and decision support systems—require different performance skills. This book examines the considerations that any database designer, administrator, or performance expert should focus their efforts on.

System performance is designed and built into a system. It does not just happen. Performance problems are usually the result of contention for, or exhaustion of, some system resource. When a system resource is exhausted, the system is unable to scale to higher levels of performance. This new performance methodology is based on careful planning and design of the database, to prevent system resources from becoming exhausted and causing down-time. By eliminating resource conflicts, systems can be made scalable to the levels required by the business.

Understanding Investment Options

With the availability of relatively inexpensive, high-powered processors, memory, and disk drives, there is a temptation to buy more system resources to improve performance. In many situations, new CPUs, memory, or more disk drives can indeed

provide an immediate performance improvement. However, any performance increases achieved by adding hardware should be considered a short-term relief to an immediate problem. If the demand and load rates on the application continue to grow, then the chance that you will face the same problem in the near future is very likely.

In other situations, additional hardware does not improve the system's performance at all. Poorly designed systems perform poorly no matter how much extra hardware is allocated. Before purchasing additional hardware, make sure that there is no serialization or single threading going on within the application. Long-term, it is generally more valuable to increase the efficiency of your application in terms of the number of physical resources used for each business transaction.

Understanding Scalability

The word *scalability* is used in many contexts in development environments. The following section provides an explanation of scalability that is aimed at application designers and performance specialists.

This section covers the following topics:

What is Scalability?

Scalability is a system's ability to process more workload, with a proportional increase in system resource usage. In other words, in a scalable system, if you double the workload, then the system would use twice as many system resources. This sounds obvious, but due to conflicts within the system, the resource usage might exceed twice the original workload.

Examples of bad scalability due to resource conflicts include the following:

- Applications requiring significant concurrency management as user populations increase
- Increased locking activities
- Increased data consistency workload
- Increased operating system workload
- Transactions requiring increases in data access as data volumes increase
- Poor SQL and index design resulting in a higher number of logical I/Os for the same number of rows returned
- Reduced availability, because database objects take longer to maintain

An application is said to be unscalable if it exhausts a system resource to the point where no more throughput is possible when it's workload is increased. Such applications result in fixed throughputs and poor response times.

Examples of resource exhaustion include the following:

- Hardware exhaustion
- Table scans in high-volume transactions causing inevitable disk I/O shortages
- Excessive network requests, resulting in network and scheduling bottlenecks

- Memory allocation causing paging and swapping
- Excessive process and thread allocation causing operating system thrashing

This means that application designers must create a design that uses the same resources, regardless of user populations and data volumes, and does not put loads on the system resources beyond their limits.

System Scalability

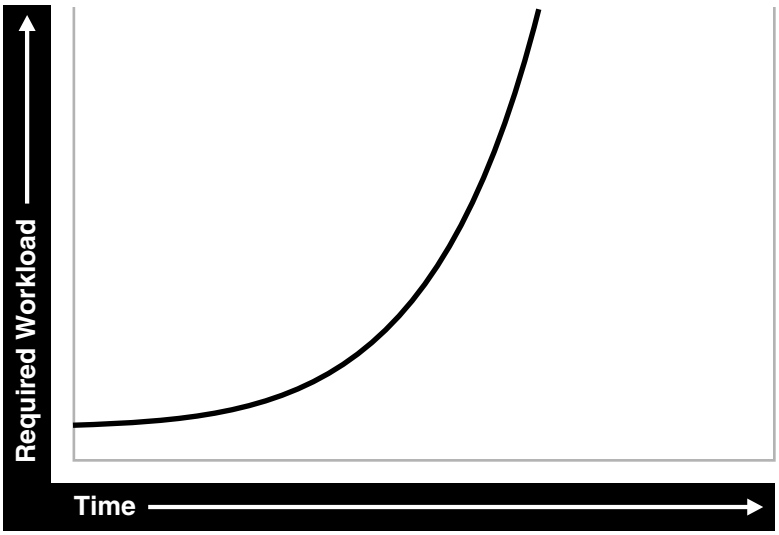
Applications that are accessible through the Internet have more complex performance and availability requirements. Some applications are designed and written only for Internet use, but even typical back-office applications—such as a general ledger application—might require some or all data to be available online.

Characteristics of Internet age applications include the following:

- Availability 24 hours a day, 365 days a year
- Unpredictable and imprecise number of concurrent users
- Difficulty in capacity planning
- Availability for any type of query
- Multitier architectures
- Stateless middleware
- Rapid development timescale
- Minimal time for testing

Figure 2–1 illustrates the classic workload growth curve, with demand growing at an increasing rate. Applications must scale with the increase of workload and also when additional hardware is added to support increasing demand. Design errors can cause the implementation to reach its maximum, regardless of additional hardware resources or re-design efforts.

Figure 2–1 Workload Growth Curve



Applications are challenged by very short development timeframes with limited time for testing and evaluation. However, bad design generally means that at some point in

the future, the system will need to be re-architected or re-implemented. If an application with known architectural and implementation limitations is deployed on the Internet, and if the workload exceeds the anticipated demand, then there is real chance of failure in the future. From a business perspective, poor performance can mean a loss of customers. If Web users do not get a response in seven seconds, then the user's attention could be lost forever.

In many cases, the cost of re-designing a system with the associated downtime costs in migrating to new implementations exceeds the costs of properly building the original system. The moral of the story is simple: design and implement with scalability in mind from the start.

Factors Preventing Scalability

When building applications, designers and architects should aim for as close to perfect scalability as possible. This is sometimes called *linear* scalability, where system throughput is directly proportional to the number of CPUs.

In real life, linear scalability is impossible for reasons beyond a designer's control. However, making the application design and implementation as scalable as possible should ensure that current and future performance objectives can be achieved through expansion of hardware components and the evolution of CPU technology.

Factors that may prevent linear scalability include:

- Poor application design, implementation, and configuration

  The application has the biggest impact on scalability. For example:

  - Poor schema design can cause expensive SQL that do not scale.
  - Poor transaction design can cause locking and serialization problems.
  - Poor connection management can cause poor response times and unreliable systems.

  However, the design is not the only problem. The physical implementation of the application can be the weak link. For example:

  - Systems can move to production environments with bad I/O strategies.
  - The production environment could use different execution plans than those generated in testing.
  - Memory-intensive applications that allocate a large amount of memory without much thought for freeing the memory at runtime can cause excessive memory usage.
  - Inefficient memory usage and memory leaks put a high stress on the operating virtual memory subsystem. This impacts performance and availability.

- Incorrect sizing of hardware components

  Bad capacity planning of all hardware components is becoming less of a problem as relative hardware prices decrease. However, too much capacity can mask scalability problems as the workload is increased on a system.

- Limitations of software components

  All software components have scalability and resource usage limitations. This applies to application servers, database servers, and operating systems. Application design should not place demands on the software beyond what it can handle.

- Limitations of Hardware Components

  Hardware is not perfectly scalable. Most multiprocessor machines can get close to linear scaling with a finite number of CPUs, but after a certain point each additional CPU can increase performance overall, but not proportionately. There might come a time when an additional CPU offers no increase in performance, or even degrades performance. This behavior is very closely linked to the workload and the operating system setup.

  Note: These factors are based on Oracle Server Performance group's experience of tuning unscalable systems.

System Architecture

There are two main parts to a system's architecture:

- Hardware and Software Components
- Configuring the Right System Architecture for Your Requirements

Hardware and Software Components

This section discusses hardware and software components.

Hardware Components

Today's designers and architects are responsible for sizing and capacity planning of hardware at each tier in a multitier environment. It is the architect's responsibility to achieve a balanced design. This is analogous to a bridge designer who must consider all the various payload and structural requirements for the bridge. A bridge is only as strong as its weakest component. As a result, a bridge is designed in balance, such that all components reach their design limits simultaneously.

The main hardware components include:

- CPU
- Memory
- I/O Subsystem
- Network

CPU There can be one or more CPUs, and they can vary in processing power from simple CPUs found in hand-held devices to high-powered server CPUs. Sizing of other hardware components is usually a multiple of the CPUs on the system. See Chapter 9, "Understanding Operating System Resources".

Memory Database and application servers require considerable amounts of memory to cache data and avoid time-consuming disk access. See Chapter 7, "Memory Configuration and Use".

I/O Subsystem The I/O subsystem can vary between the hard disk on a client PC and high performance disk arrays. Disk arrays can perform thousands of I/Os each second and provide availability through redundancy in terms of multiple I/O paths and hot pluggable mirrored disks. See Chapter 8, "I/O Configuration and Design".

Network All computers in a system are connected to a network, from a modem line to a high speed internal LAN. The primary concerns with network specifications are bandwidth (volume) and latency (speed).

Software Components

The same way computers have common hardware components, applications have common functional components. By dividing software development into functional components, it is possible to better comprehend the application design and architecture. Some components of the system are performed by existing software bought to accelerate application implementation, or to avoid re-development of common components.

The difference between software components and hardware components is that while hardware components only perform one task, a piece of software can perform the roles of various software components. For example, a disk drive only stores and retrieves data, but a client program can manage the user interface and perform business logic.

Most applications involve the following components:

- Managing the User Interface
- Implementing Business Logic
- Managing User Requests and Resource Allocation
- Managing Data and Transactions

Managing the User Interface This component is the most visible to application users. This includes the following functions:

- Painting the screen in front of the user
- Collecting user data and transferring it to business logic
- Validating data entry
- Navigating through levels or states of the application

Implementing Business Logic This component implements core business rules that are central to the application function. Errors made in this component can be very costly to repair. This component is implemented by a mixture of declarative and procedural approaches. An example of a declarative activity is defining unique and foreign keys. An example of procedure-based logic is implementing a discounting strategy.

Common functions of this component include:

- Moving a data model to a relational table structure
- Defining constraints in the relational table structure
- Coding procedural logic to implement business rules

Managing User Requests and Resource Allocation This component is implemented in all pieces of software. However, there are some requests and resources that can be influenced by the application design and some that cannot.

In a multiuser application, most resource allocation by user requests are handled by the database server or the operating system. However, in a large application where the number of users and their usage pattern is unknown or growing rapidly, the system architect must be proactive to ensure that no single software component becomes overloaded and unstable.

Common functions of this component include:

- Connection management with the database
- Executing SQL efficiently (cursors and SQL sharing)
- Managing client state information
- Balancing the load of user requests across hardware resources
- Setting operational targets for hardware/software components
- Persistent queuing for asynchronous execution of tasks

Managing Data and Transactions This component is largely the responsibility of the database server and the operating system.

Common functions of this component include:

- Providing concurrent access to data using locks and transactional semantics
- Providing optimized access to the data using indexes and memory cache
- Ensuring that data changes are logged in the event of a hardware failure
- Enforcing any rules defined for the data

Configuring the Right System Architecture for Your Requirements

Configuring the initial system architecture is a largely iterative process. Architects must satisfy the system requirements within budget and schedule constraints. If the system requires interactive users transacting business-making decisions based on the contents of a database, then user requirements drive the architecture. If there are few interactive users on the system, then the architecture is process-driven.

Examples of interactive user applications:

- Accounting and bookkeeping applications
- Order entry systems
- Email servers
- Web-based retail applications
- Trading systems

Examples of process-driven applications:

- Utility billing systems
- Fraud detection systems
- Direct mail

In many ways, process-driven applications are easier to design than multiuser applications because the user interface element is eliminated. However, because the objectives are process-oriented, architects not accustomed to dealing with large data volumes and different success factors can become confused. Process-driven applications draw from the skills sets used in both user-based applications and data warehousing. Therefore, this book focuses on evolving system architectures for interactive users.

Note: Generating a system architecture is not a deterministic process. It requires careful consideration of business requirements, technology choices, existing infrastructure and systems, and actual physical resources, such as budget and manpower.

The following questions should stimulate thought on architecture, though they are not a definitive guide to system architecture. These questions demonstrate how business requirements can influence the architecture, ease of implementation, and overall performance and availability of a system. For example:

- How many users will the system support?

  Most applications fall into one of the following categories:

  - Very few users on a lightly-used or exclusive machine

    For this type of application, there is usually one user. The focus of the application design is to make the single user as productive as possible by providing good response time, yet make the application require minimal administration. Users of these applications rarely interfere with each other and have minimal resource conflicts.

  - A medium to large number of users in a corporation using shared applications

    For this type of application, the users are limited by the number of employees in the corporation actually transacting business through the system. Therefore, the number of users is predictable. However, delivering a reliable service is crucial to the business. The users will be using a shared resource, so design efforts must address response time under heavy system load, escalation of resource for each session usage, and room for future growth.

  - An infinite user population distributed on the Internet

    For this type of application, extra engineering effort is required to ensure that no system component exceeds its design limits. This would create a bottleneck that brings the system to a halt and becomes unstable. These applications require complex load balancing, stateless application servers, and efficient database connection management. In addition, statistics and governors should be used to ensure that the user gets some feedback if their requests cannot be satisfied due to system overload.

- What will be the user interaction method?

  The choices of user interface range from a simple Web browser to a custom client program.

- Where are the users located?

  The distance between users influences how the application is engineered to cope with network latencies. The location also affects which times of the day are busy, when it is impossible to perform batch or system maintenance functions.

- What is the network speed?

  Network speed affects the amount of data and the conversational nature of the user interface with the application and database servers. A highly conversational user interface can communicate with back-end servers on every key stroke or field level validation. A less conversational interface works on a screen-sent and a screen-received model. On a slow network, it is impossible to achieve high data entry speeds with a highly conversational user interface.

- How much data will the user access, and how much of that data is largely read only?

  The amount of data queried online influences all aspects of the design, from table and index design to the presentation layers. Design efforts must ensure that user response time is not a function of the size of the database. If the application is largely read only, then replication and data distribution to local caches in the

application servers become a viable option. This also reduces workload on the core transactional server.

- What is the user response time requirement?

  Consideration of the user type is important. If the user is an executive who requires accurate information to make split second decisions, then user response time cannot be compromised. Other types of users, such as users performing data entry activities, might not need such a high level of performance.

- Do users expect 24 hour service?

  This is mandatory for today's Internet applications where trade is conducted 24 hours a day. However, corporate systems that run in a single time zone might be able to tolerate after-hours downtime. This after-hours downtime can be used to run batch processes or to perform system administration. In this case, it might be more economic not to run a fully-available system.

- Must all changes be made in real time?

  It is important to determine if transactions need to be executed within the user response time, or if they can they be queued for asynchronous execution.

The following are secondary questions, which can also influence the design, but really have more impact on budget and ease of implementation. For example:

- How big will the database be?

  This influences the sizing of the database server machine. On systems with a very large database, it might be necessary to have a bigger machine than dictated by the workload. This is because the administration overhead with large databases is largely a function of the database size. As tables and indexes grow, it takes proportionately more CPUs to allow table reorganizations and index builds to complete in an acceptable time limit.

- What is the required throughput of business transactions?
- What are the availability requirements?
- Do skills exist to build and administer this application?
- What compromises will be forced by budget constraints?

Application Design Principles

This section describes the following design decisions that are involved in building applications:

Simplicity In Application Design

Applications are no different than any other designed and engineered product. Well-designed structures, machines, and tools are usually reliable, easy to use and maintain, and simple in concept. In the most general terms, if the design looks correct, then it probably is. This principle should always be kept in mind when building applications.

Consider some of the following design issues:

- If the table design is so complicated that nobody can fully understand it, then the table is probably poorly designed.
- If SQL statements are so long and involved that it would be impossible for any optimizer to effectively optimize it in real time, then there is probably a bad statement, underlying transaction, or table design.
- If there are indexes on a table and the same columns are repeatedly indexed, then there is probably a poor index design.
- If queries are submitted without suitable qualification for rapid response for online users, then there is probably a poor user interface or transaction design.
- If the calls to the database are abstracted away from the application logic by many layers of software, then there is probably a bad software development method.

Data Modeling

Data modeling is important to successful relational application design. This should be done in a way that quickly represents the business practices. Chances are, there will be heated debates about the correct data model. The important thing is to apply greatest modeling efforts to those entities affected by the most frequent business transactions. In the modeling phase, there is a great temptation to spend too much time modeling the non-core data elements, which results in increased development lead times. Use of modeling tools can then rapidly generate schema definitions and can be useful when a fast prototype is required.

Table and Index Design

Table design is largely a compromise between flexibility and performance of core transactions. To keep the database flexible and able to accommodate unforeseen workloads, the table design should be very similar to the data model, and it should be normalized to at least 3rd normal form. However, certain core transactions required by users can require selective denormalization for performance purposes.

Examples of this technique include storing tables pre-joined, the addition of derived columns, and aggregate values. Oracle provides numerous options for storage of aggregates and pre-joined data by clustering and materialized view functions. These features allow a simpler table design to be adopted initially.

Again, focus and resources should be spent on the business critical tables, so that optimal performance can be achieved. For non-critical tables, shortcuts in design can be adopted to enable a more rapid application development. However, if prototyping and testing a non-core table becomes a performance problem, then remedial design effort should be applied immediately.

Index design is also a largely iterative process, based on the SQL generated by application designers. However, it is possible to make a sensible start by building indexes that enforce primary key constraints and indexes on known access patterns, such as a person's name. As the application evolves and testing is performed on

realistic amounts of data, certain queries will need performance improvements for which building a better index is a good solution. The following list of indexing design ideas should be considered when building a new index:

- Appending Columns to an Index or Using Index-Organized Tables
- Using a Different Index Type
- Finding the Cost of an Index
- Serializing within Indexes
- Ordering Columns in an Index

Appending Columns to an Index or Using Index-Organized Tables

One of the easiest ways to speed up a query is to reduce the number of logical I/Os by eliminating a table access from the execution plan. This can be done by appending to the index all columns referenced by the query. These columns are the select list columns, and any required join or sort columns. This technique is particularly useful in speeding up online applications response times when time-consuming I/Os are reduced. This is best applied when testing the application with properly sized data for the first time.

The most aggressive form of this technique is to build an index-organized table (IOT). However, you must be careful that the increased leaf size of an IOT does not undermine the efforts to reduce I/O.

Using a Different Index Type

There are several index types available, and each index has benefits for certain situations. The following list gives performance ideas associated with each index type.

B-Tree Indexes These indexes are the standard index type, and they are excellent for primary key and highly-selective indexes. Used as concatenated indexes, B-tree indexes can be used to retrieve data sorted by the index columns.

Bitmap Indexes These indexes are suitable for low cardinality data. Through compression techniques, they can generate a large number of rowids with minimal I/O. Combining bitmap indexes on non-selective columns allows efficient `AND` and `OR` operations with a great number of rowids with minimal I/O. Bitmap indexes are particularly efficient in queries with `COUNT`(), because the query can be satisfied within the index.

Function-based Indexes These indexes allow access through a B-tree on a value derived from a function on the base data. Function-based indexes have some limitations with regards to the use of nulls, and they require that you have the query optimizer enabled.

Function-based indexes are particularly useful when querying on composite columns to produce a derived result or to overcome limitations in the way data is stored in the database. An example is querying for line items in an order exceeding a certain value derived from (sales price - discount) x quantity, where these were columns in the table. Another example is to apply the `UPPER` function to the data to allow case-insensitive searches.

Partitioned Indexes Partitioning a global index allows partition pruning to take place within an index access, which results in reduced I/Os. By definition of good range or list partitioning, fast index scans of the correct index partitions can result in very fast query times.

Reverse Key Indexes These indexes are designed to eliminate index hot spots on insert applications. These indexes are excellent for insert performance, but they are limited in that they cannot be used for index range scans.

Finding the Cost of an Index

Building and maintaining an index structure can be expensive, and it can consume resources such as disk space, CPU, and I/O capacity. Designers must ensure that the benefits of any index outweigh the negatives of index maintenance.

Use this simple estimation guide for the cost of index maintenance: each index maintained by an `INSERT`, `DELETE`, or `UPDATE` of the indexed keys requires about three times as much resource as the actual DML operation on the table. What this means is that if you `INSERT` into a table with three indexes, then it will be approximately 10 times slower than an `INSERT` into a table with no indexes. For DML, and particularly for `INSERT`-heavy applications, the index design should be seriously reviewed, which might require a compromise between the query and `INSERT` performance.

> **See Also:** *Oracle Database Administrator's Guide* for information on monitoring index usage

Serializing within Indexes

Use of sequences, or timestamps, to generate key values that are indexed themselves can lead to database hotspot problems, which affect response time and throughput. This is usually the result of a monotonically growing key that results in a right-growing index. To avoid this problem, try to generate keys that insert over the full range of the index. This results in a well-balanced index that is more scalable and space efficient. You can achieve this by using a reverse key index or using a cycling sequence to prefix and sequence values.

Ordering Columns in an Index

Designers should be flexible in defining any rules for index building. Depending on your circumstances, use one of the following two ways to order the keys in an index:

- Order columns with most selectivity first. This method is the most commonly used, because it provides the fastest access with minimal I/O to the actual rowids required. This technique is used mainly for primary keys and for very selective range scans.
- Order columns to reduce I/O by clustering or sorting data. In large range scans, I/Os can usually be reduced by ordering the columns in the least selective order, or in a manner that sorts the data in the way it should be retrieved. See Chapter 14, "Using Indexes and Clusters".

Using Views

Views can speed up and simplify application design. A simple view definition can mask data model complexity from the programmers whose priorities are to retrieve, display, collect, and store data.

However, while views provide clean programming interfaces, they can cause sub-optimal, resource-intensive queries. The worst type of view use is when a view references other views, and when they are joined in queries. In many cases, developers can satisfy the query directly from the table without using a view. Usually, because of their inherent properties, views make it difficult for the optimizer to generate the optimal execution plan.

SQL Execution Efficiency

In the design and architecture phase of any system development, care should be taken to ensure that the application developers understand SQL execution efficiency. To do this, the development environment must support the following characteristics:

- Good Database Connection Management

  Connecting to the database is an expensive operation that is highly unscalable. Therefore, the number of concurrent connections to the database should be minimized as much as possible. A simple system, where a user connects at application initialization, is ideal. However, in a Web-based or multitiered application, where application servers are used to multiplex database connections to users, this can be difficult. With these types of applications, design efforts should ensure that database connections are pooled and are not reestablished for each user request.

- Good Cursor Usage and Management

  Maintaining user connections is equally important to minimizing the parsing activity on the system. Parsing is the process of interpreting a SQL statement and creating an execution plan for it. This process has many phases, including syntax checking, security checking, execution plan generation, and loading shared structures into the shared pool. There are two types of parse operations:

  - Hard Parsing: A SQL statement is submitted for the first time, and no match is found in the shared pool. Hard parses are the most resource-intensive and unscalable, because they perform all the operations involved in a parse.
  - Soft Parsing: A SQL statement is submitted for the first time, and a match *is* found in the shared pool. The match can be the result of previous execution by another user. The SQL statement is shared, which is good for performance. However, soft parses are not ideal, because they still require syntax and security checking, which consume system resources.

  Since parsing should be minimized as much as possible, application developers should design their applications to parse SQL statements once and execute them many times. This is done through cursors. Experienced SQL programmers should be familiar with the concept of opening and re-executing cursors.

  Application developers must also ensure that SQL statements are shared within the shared pool. To do this, bind variables to represent the parts of the query that change from execution to execution. If this is not done, then the SQL statement is likely to be parsed once and never re-used by other users. To ensure that SQL is shared, use bind variables and do not use string literals with SQL statements. For example:

  Statement with string literals:

```
SELECT * FROM employees
  WHERE last_name LIKE 'KING';
```

  Statement with bind variables:

```
SELECT * FROM employees
  WHERE last_name LIKE :1;
```

  The following example shows the results of some tests on a simple OLTP application:

```
Test                          #Users Supported
No Parsing all statements           270
```

```
Soft Parsing all statements         150
Hard Parsing all statements          60
Re-Connecting for each Transaction   30
```

These tests were performed on a four-CPU machine. The differences increase as the number of CPUs on the system increase. See Chapter 16, "SQL Tuning Overview" for information on optimizing SQL statements.

Implementing the Application

The choice of development environment and programming language is largely a function of the skills available in the development team and architectural decisions made when specifying the application. There are, however, some simple performance management rules that can lead to scalable, high-performance applications.

1. Choose a development environment suitable for software components, and do not let it limit your design for performance decisions. If it does, then you probably chose the wrong language or environment.
   - User Interface

     The programming model can vary between HTML generation and calling the windowing system directly. The development method should focus on response time of the user interface code. If HTML or Java is being sent over a network, then try to minimize network volume and interactions.
   - Business Logic

     Interpreted languages, such as Java and PL/SQL, are ideal to encode business logic. They are fully portable, which makes upgrading logic relatively easy. Both languages are syntactically rich to allow code that is easy to read and interpret. If business logic requires complex mathematical functions, then a compiled binary language might be needed. The business logic code can be on the client machine, the application server, and the database server. However, the application server is the most common location for business logic.
   - User Requests and Resource Allocation

     Most of this is not affected by the programming language, but tools and 4th generation languages that mask database connection and cursor management might use inefficient mechanisms. When evaluating these tools and environments, check their database connection model and their use of cursors and bind variables.
   - Data Management and Transactions

     Most of this is not affected by the programming language.
2. When implementing a software component, implement its function and not the functionality associated with other components. Implementing another component's functionality results in sub-optimal designs and implementations. This applies to all components.
3. Do not leave gaps in functionality or have software components under-researched in design, implementation, or testing. In many cases, gaps are not discovered until the application is rolled out or tested at realistic volumes. This is usually a sign of poor architecture or initial system specification. Data archival/purge modules are most frequently neglected during initial system design, build, and implementation.
4. When implementing procedural logic, implement in a procedural language, such as C, Java, or PL/SQL. When implementing data access (queries) or data changes

(DML), use SQL. This rule is specific to the business logic modules of code where procedural code is mixed with data access (non-procedural SQL) code. There is great temptation to put procedural logic into the SQL access. This tends to result in poor SQL that is resource-intensive. SQL statements with `DECODE` case statements are very often candidates for optimization, as are statements with a large amount of `OR` predicates or set operators, such as `UNION` and `MINUS`.

5. Cache frequently accessed, rarely changing data that is expensive to retrieve on a repeated basis. However, make this cache mechanism easy to use, and ensure that it is indeed cheaper than accessing the data in the original method. This is applicable to all modules where frequently used data values should be cached or stored locally, rather than be repeatedly retrieved from a remote or expensive data store.

   The most common examples of candidates for local caching include the following:

   - Today's date. `SELECT SYSDATE FROM DUAL` can account for over 60% of the workload on a database.
   - The current user name.
   - Repeated application variables and constants, such as tax rates, discounting rates, or location information.
   - Caching data locally can be further extended into building a local data cache into the application server middle tiers. This helps take load off the central database servers. However, care should be taken when constructing local caches so that they do not become so complex that they cease to give a performance gain.
   - Local sequence generation.

   The design implications of using a cache should be considered. For example, if a user is connected at midnight and the date is cached, then the user's date value becomes invalid.

6. Optimize the interfaces between components, and ensure that all components are used in the most scalable configuration. This rule requires minimal explanation and applies to all modules and their interfaces.

7. Use foreign key references. Enforcing referential integrity through an application is expensive. You can maintain a foreign key reference by selecting the column value of the child from the parent and ensuring that it exists. The foreign key constraint enforcement supplied by Oracle—which does not use SQL, is fast, easy to declare—and does not create network traffic.

8. Consider setting up action and module names in the application to use with End to End Application Tracing. This allows greater flexibility in tracing workload problems. See "End to End Application Tracing" on page 21-1.

Trends in Application Development

The two biggest challenges in application development today are the increased use of Java to replace compiled C or C++ applications, and increased use of object-oriented techniques, influencing the schema design.

Java provides better portability of code and availability to programmers. However, there are a number of performance implications associated with Java. Since Java is an interpreted language, it is slower at executing similar logic than compiled languages such as C. As a result, resource usage of client machines increases. This requires more

powerful CPUs to be applied in the client or middle-tier machines and greater care from programmers to produce efficient code.

Since Java is an object-oriented language, it encourages insulation of data access into classes not performing the business logic. As a result, programmers might invoke methods without knowledge of the efficiency of the data access method being used. This tends to result in database access that is very minimal and uses the simplest and crudest interfaces to the database.

With this type of software design, queries do not always include all the `WHERE` predicates to be efficient, and row filtering is performed in the Java program. This is very inefficient. In addition, for DML operations—and especially for `INSERT`s—single `INSERT`s are performed, making use of the array interface impossible. In some cases, this is made more inefficient by procedure calls. More resources are used moving the data to and from the database than in the actual database calls.

In general, it is best to place data access calls next to the business logic to achieve the best overall transaction design.

The acceptance of object-orientation at a programming level has led to the creation of object-oriented databases within the Oracle Server. This has manifested itself in many ways, from storing object structures within `BLOB`s and only using the database effectively as an indexed card file to the use of the Oracle object relational features.

If you adopt an object-oriented approach to schema design, then make sure that you do not lose the flexibility of the relational storage model. In many cases, the object-oriented approach to schema design ends up in a heavily denormalized data structure that requires considerable maintenance and `REF` pointers associated with objects. Often, these designs represent a step backward to the hierarchical and network database designs that were replaced with the relational storage method.

In summary, if you are storing your data in your database for the long-term and anticipate a degree of ad hoc queries or application development on the same schema, then you will probably find that the relational storage method gives the best performance and flexibility.

Workload Testing, Modeling, and Implementation

This section describes workload estimation, modeling, implementation, and testing. This section covers the following topics:

- Sizing Data
- Estimating Workloads
- Application Modeling
- Testing, Debugging, and Validating a Design

Sizing Data

You could experience errors in your sizing estimates when dealing with variable length data if you work with a poor sample set. As data volumes grow, your key lengths could grow considerably, altering your assumptions for column sizes.

When the system becomes operational, it becomes more difficult to predict database growth, especially for indexes. Tables grow over time, and indexes are subject to the individual behavior of the application in terms of key generation, insertion pattern, and deletion of rows. The worst case is where you insert using an ascending key, and then delete most rows from the left-hand side but not all the rows. This leaves gaps

and wasted space. If you have index use like this, make sure that you know how to use the online index rebuild facility.

DBAs should monitor space allocation for each object and look for objects that may grow out of control. A good understanding of the application can highlight objects that may grow rapidly or unpredictably. This is a crucial part of both performance and availability planning for any system. When implementing the production database, the design should attempt to ensure that minimal space management takes place when interactive users are using the application. This applies for all data, temp, and rollback segments.

Estimating Workloads

Estimation of workloads for capacity planning and testing purposes is often described as a black art. When considering the number of variables involved it is easy to see why this process is largely impossible to get precisely correct. However, designers need to specify machines with CPUs, memory, and disk drives, and eventually roll out an application. There are a number of techniques used for sizing, and each technique has merit. When sizing, it is best to use the following two methods to validate your decision-making process and provide supporting documentation:

- Extrapolating From a Similar System
- Benchmarking

Extrapolating From a Similar System

This is an entirely empirical approach where an existing system of similar characteristics and known performance is used as a basis system. The specification of this system is then modified by the sizing specialist according to the known differences. This approach has merit in that it correlates with an existing system, but it provides little assistance when dealing with the differences.

This approach is used in nearly all large engineering disciplines when preparing the cost of an engineering project, such as a large building, a ship, a bridge, or an oil rig. If the reference system is an order of magnitude different in size from the anticipated system, then some of the components may have exceeded their design limits.

Benchmarking

The benchmarking process is both resource and time consuming, and it might not produce the correct results. By simulating an application in early development or prototype form, there is a danger of measuring something that has no resemblance to the actual production system. This sounds strange, but over the many years of benchmarking customer applications with the database development organization, Oracle has yet to see reliable correlation between the benchmark application and the actual production system. This is mainly due to the number of application inefficiencies introduced in the development process.

However, benchmarks have been used successfully to size systems to an acceptable level of accuracy. In particular, benchmarks are very good at determining the actual I/O requirements and testing recovery processes when a system is fully loaded.

Benchmarks by their nature stress all system components to their limits. As all components are being stressed, be prepared to see all errors in application design and implementation manifest themselves while benchmarking. Benchmarks also test database, operating system, and hardware components. Since most benchmarks are performed in a rush, expect setbacks and problems when a system component fails.

Benchmarking is a stressful activity, and it takes considerable experience to get the most out of a benchmarking exercise.

Application Modeling

Modeling the application can range from complex mathematical modeling exercises to the classic simple calculations performed on the back of an envelope. Both methods have merit, with one attempting to be very precise and the other making gross estimates. The downside of both methods is that they do not allow for implementation errors and inefficiencies.

The estimation and sizing process is an imprecise science. However, by investigating the process, some intelligent estimates can be made. The whole estimation process makes no allowances for application inefficiencies introduced by poor SQL, index design, or cursor management. A sizing engineer should build in margin for application inefficiencies. A performance engineer should discover the inefficiencies and makes the estimates look realistic. The process of discovering the application inefficiencies is described in the Oracle performance method.

Testing, Debugging, and Validating a Design

The testing process mainly consists of functional and stability testing. At some point in the process, performance testing is performed.

The following list describes some simple rules for performance testing an application. If correctly documented, this provides important information for the production application and the capacity planning process after the application has gone live.

- Use the Automatic Database Diagnostic Monitor (ADDM) and SQL Tuning Advisor for design validation
- Test with realistic data volumes and distributions

  All testing must be done with fully populated tables. The test database should contain data representative of the production system in terms of data volume and cardinality between tables. All the production indexes should be built and the schema statistics should be populated correctly.
- Use the correct optimizer mode

  All testing should be performed with the optimizer mode that will be used in production. All Oracle research and development effort is focused upon the query optimizer, and therefore the use of the query optimizer is recommended.
- Test a single user performance

  A single user on an idle or lightly-used system should be tested for acceptable performance. If a single user cannot achieve acceptable performance under ideal conditions, it is impossible there will be acceptable performance under multiple users where resources are shared.
- Obtain and document plans for all SQL statements

  Obtain an execution plan for each SQL statement, and some metrics should be obtained for at least one execution of the statement. This process should be used to validate that an optimal execution plan is being obtained by the optimizer, and the relative cost of the SQL statement is understood in terms of CPU time and physical I/Os. This process assists in identifying the heavy use transactions that will require the most tuning and performance work in the future.
- Attempt multiuser testing

This process is difficult to perform accurately, because user workload and profiles might not be fully quantified. However, transactions performing DML statements should be tested to ensure that there are no locking conflicts or serialization problems.

- Test with the correct hardware configuration

  It is important to test with a configuration as close to the production system as possible. This is particularly important with respect to network latencies, I/O sub-system bandwidth, and processor type and speed. A failure to do this could result in an incorrect analysis of potential performance problems.

- Measure steady state performance

  When benchmarking, it is important to measure the performance under steady state conditions. Each benchmark run should have a ramp-up phase, where users are connected to the application and gradually start performing work on the application. This process allows for frequently cached data to be initialized into the cache and single execution operations—such as parsing—to be completed prior to the steady state condition. Likewise, at the end of a benchmark run, there should be a ramp-down period, where resources are freed from the system and users cease work and disconnect.

Deploying New Applications

This section describes the following design decisions involved in deploying applications:

- Rollout Strategies
- Performance Checklist

Rollout Strategies

When new applications are rolled out, two strategies are commonly adopted:

- Big Bang approach - all users migrate to the new system at once
- Trickle approach - users slowly migrate from existing systems to the new one

Both approaches have merits and disadvantages. The Big Bang approach relies on reliable testing of the application at the required scale, but has the advantage of minimal data conversion and synchronization with the old system, because it is simply switched off. The Trickle approach allows debugging of scalability issues as the workload increases, but might mean that data needs to be migrated to and from legacy systems as the transition takes place.

It is hard to recommend one approach over the other, because each method has associated risks that could lead to system outages as the transition takes place. Certainly, the Trickle approach allows profiling of real users as they are introduced to the new application, and allows the system to be reconfigured while only affecting the migrated users. This approach affects the work of the early adopters, but limits the load on support services. This means that unscheduled outages only affect a small percentage of the user population.

The decision on how to roll out a new application is specific to each business. The approach adopted will have its own unique pressures and stresses. The more testing and knowledge derived from the testing process, the more you will realize what is best for the rollout.

Performance Checklist

To assist in the rollout process, build a list of tasks that—if performed correctly—will increase the chance of optimal performance in production and—if there is a problem—enable rapid debugging of the application:

1. When you create the control file for the production database, allow for growth by setting `MAXINSTANCES`, `MAXDATAFILES`, `MAXLOGFILES`, `MAXLOGMEMBERS`, and `MAXLOGHISTORY` to values higher than what you anticipate for the rollout. This results in more disk space usage and bigger control files, but saves time later should these need extension in an emergency.
2. Set block size to the value used to develop the application. Export the schema statistics from the development/test environment to the production database if the testing was done on representative data volumes and the current SQL execution plans are correct.
3. Set the minimal number of initialization parameters. Ideally, most other parameters should be left at default. If there is more tuning to perform, this shows up when the system is under load. See Chapter 4, "Configuring a Database for Performance" for information on parameter settings in an initial instance configuration.
4. Be prepared to manage block contention by setting storage options of database objects. Tables and indexes that experience high `INSERT/UPDATE/DELETE` rates should be created with automatic segment space management. To avoid contention of rollback segments, automatic undo management should be used. See Chapter 4, "Configuring a Database for Performance" for information on undo and temporary segments.
5. All SQL statements should be verified to be optimal and their resource usage understood.
6. Validate that middleware and programs that connect to the database are efficient in their connection management and do not logon/logoff repeatedly.
7. Validate that the SQL statements use cursors efficiently. Each SQL statement should be parsed once and then executed multiple times. The most common reason this does not happen is because bind variables are not used properly and `WHERE` clause predicates are sent as string literals. If the precompilers are used to develop the application, then make sure that the parameters `MAXOPENCURSORS`, `HOLD_CURSOR`, and `RELEASE_CURSOR` have been reset from the default values prior to precompiling the application.
8. Validate that all schema objects have been correctly migrated from the development environment to the production database. This includes tables, indexes, sequences, triggers, packages, procedures, functions, Java objects, synonyms, grants, and views. Ensure that any modifications made in testing are made to the production system.
9. As soon as the system is rolled out, establish a baseline set of statistics from the database and operating system. This first set of statistics validates or corrects any assumptions made in the design and rollout process.
10. Start anticipating the first bottleneck (there will always be one) and follow the Oracle performance method to make performance improvement. For more information, see Chapter 3, "Performance Improvement Methods".

3

Performance Improvement Methods

This chapter discusses Oracle improvement methods and contains the following sections:

The Oracle Performance Improvement Method

Oracle performance methodology helps you to pinpoint performance problems in your Oracle system. This involves identifying bottlenecks and fixing them. It is recommended that changes be made to a system only after you have confirmed that there is a bottleneck.

Performance improvement, by its nature, is iterative. For this reason, removing the first bottleneck might not lead to performance improvement immediately, because another bottleneck might be revealed. Also, in some cases, if serialization points move to a more inefficient sharing mechanism, then performance could degrade. With experience, and by following a rigorous method of bottleneck elimination, applications can be debugged and made scalable.

Performance problems generally result from either a lack of throughput, unacceptable user/job response time, or both. The problem might be localized between application modules, or it might be for the entire system.

Before looking at any database or operating system statistics, it is crucial to get feedback from the most important components of the system: the users of the system and the people ultimately paying for the application. Typical user feedback includes statements like the following:

- "The online performance is so bad that it prevents my staff from doing their jobs."
- "The billing run takes too long."
- "When I experience high amounts of Web traffic, the response time becomes unacceptable, and I am losing customers."
- "I am currently performing 5000 trades a day, and the system is maxed out. Next month, we roll out to all our users, and the number of trades is expected to quadruple."

From candid feedback, it is easy to set critical success factors for any performance work. Determining the performance targets and the performance engineer's exit criteria make managing the performance process much simpler and more successful at all levels. These critical success factors are better defined in terms of real business goals rather than system statistics.

Some real business goals for these typical user statements might be:

- "The billing run must process 1,000,000 accounts in a three-hour window."
- "At a peak period on a Web site, the response time will not exceed five seconds for a page refresh."
- "The system must be able to process 25,000 trades in an eight-hour window."

The ultimate measure of success is the user's perception of system performance. The performance engineer's role is to eliminate any bottlenecks that degrade performance. These bottlenecks could be caused by inefficient use of limited shared resources or by abuse of shared resources, causing serialization. Because all shared resources are limited, the goal of a performance engineer is to maximize the number of business operations with efficient use of shared resources. At a very high level, the entire database server can be seen as a shared resource. Conversely, at a low level, a single CPU or disk can be seen as shared resources.

The Oracle performance improvement method can be applied until performance goals are met or deemed impossible. This process is highly iterative, and it is inevitable that some investigations will be made that have little impact on the performance of the system. It takes time and experience to develop the necessary skills to accurately pinpoint critical bottlenecks in a timely manner. However, prior experience can sometimes work against the experienced engineer who neglects to use the data and statistics available to him. It is this type of behavior that encourages database tuning by myth and folklore. This is a very risky, expensive, and unlikely to succeed method of database tuning.

The Automatic Database Diagnostic Monitor (ADDM) implements parts of the performance improvement method and analyzes statistics to provide automatic diagnosis of major performance issues. Using ADDM can significantly shorten the time required to improve the performance of a system. See Chapter 6, "Automatic Performance Diagnostics" for a description of ADDM.

Today's systems are so different and complex that hard and fast rules for performance analysis cannot be made. In essence, the Oracle performance improvement method defines a way of working, but not a definitive set of rules. With bottleneck detection, the only rule is that there are no rules! The best performance engineers use the data provided and think laterally to determine performance problems.

Steps in The Oracle Performance Improvement Method

1. Perform the following initial standard checks:
   a. Get candid feedback from users. Determine the performance project's scope and subsequent performance goals, as well as performance goals for the future. This process is key in future capacity planning.
   b. Get a full set of operating system, database, and application statistics from the system when the performance is both good and bad. If these are not available, then get whatever is available. Missing statistics are analogous to missing evidence at a crime scene: They make detectives work harder and it is more time-consuming.
   c. Sanity-check the operating systems of all machines involved with user performance. By sanity-checking the operating system, you look for hardware or operating system resources that are fully utilized. List any over-used resources as symptoms for analysis later. In addition, check that all hardware shows no errors or diagnostics.

2. Check for the top ten most common mistakes with Oracle, and determine if any of these are likely to be the problem. List these as symptoms for later analysis. These are included because they represent the most likely problems. ADDM automatically detects and reports nine of these top ten issues. See Chapter 6, "Automatic Performance Diagnostics" and "Top Ten Mistakes Found in Oracle Systems" on page 3-4.
3. Build a conceptual model of what is happening on the system using the symptoms as clues to understand what caused the performance problems. See "A Sample Decision Process for Performance Conceptual Modeling" on page 3-3.
4. Propose a series of remedy actions and the anticipated behavior to the system, then apply them in the order that can benefit the application the most. ADDM produces recommendations each with an expected benefit. A golden rule in performance work is that you only change one thing at a time and then measure the differences. Unfortunately, system downtime requirements might prohibit such a rigorous investigation method. If multiple changes are applied at the same time, then try to ensure that they are isolated so that the effects of each change can be independently validated.
5. Validate that the changes made have had the desired effect, and see if the user's perception of performance has improved. Otherwise, look for more bottlenecks, and continue refining the conceptual model until your understanding of the application becomes more accurate.
6. Repeat the last three steps until performance goals are met or become impossible due to other constraints.

This method identifies the biggest bottleneck and uses an objective approach to performance improvement. The focus is on making large performance improvements by increasing application efficiency and eliminating resource shortages and bottlenecks. In this process, it is anticipated that minimal (less than 10%) performance gains are made from instance tuning, and large gains (100% +) are made from isolating application inefficiencies.

A Sample Decision Process for Performance Conceptual Modeling

Conceptual modeling is almost deterministic. However, as your performance tuning experience increases, you will appreciate that there are no real rules to follow. A flexible heads-up approach is required to interpret the various statistics and make good decisions.

For a quick and easy approach to performance tuning, use the Automatic Database Diagnostic Monitor (ADDM). ADDM automatically monitors your Oracle system and provides recommendations for solving performance problems should problems occur. For example, suppose a DBA receives a call from a user complaining that the system is slow. The DBA simply examines the latest ADDM report to see which of the recommendations should be implemented to solve the problem. See Chapter 6, "Automatic Performance Diagnostics" for information on the features that help monitor and diagnose Oracle systems.

The following steps illustrate how a performance engineer might look for bottlenecks without using automatic diagnostic features. These steps are only intended as a guideline for the manual process. With experience, performance engineers add to the steps involved. This analysis assumes that statistics for both the operating system and the database have been gathered.

1. Is the response time/batch run time acceptable for a single user on an empty or lightly loaded machine?

If it is not acceptable, then the application is probably not coded or designed optimally, and it will never be acceptable in a multiple user situation when system resources are shared. In this case, get application internal statistics, and get SQL Trace and SQL plan information. Work with developers to investigate problems in data, index, transaction SQL design, and potential deferral of work to batch/background processing.

2. Is all the CPU being utilized?

   If the kernel utilization is over 40%, then investigate the operating system for network transfers, paging, swapping, or process thrashing. Otherwise, move onto CPU utilization in user space. Check to see if there are any non-database jobs consuming CPU on the machine limiting the amount of shared CPU resources, such as backups, file transforms, print queues, and so on. After determining that the database is using most of the CPU, investigate the top SQL by CPU utilization. These statements form the basis of all future analysis. Check the SQL and the transactions submitting the SQL for optimal execution. Oracle provides CPU statistics in `V$SQL` and `V$SQLSTATS`.

   See Also: *Oracle Database Reference* for more information on `V$SQL` and V$SQLSTATS

   If the application is optimal and there are no inefficiencies in the SQL execution, consider rescheduling some work to off-peak hours or using a bigger machine.

3. At this point, the system performance is unsatisfactory, yet the CPU resources are not fully utilized.

   In this case, you have serialization and unscalable behavior within the server. Get the `WAIT_EVENTS` statistics from the server, and determine the biggest serialization point. If there are no serialization points, then the problem is most likely outside the database, and this should be the focus of investigation. Elimination of `WAIT_EVENTS` involves modifying application SQL and tuning database parameters. This process is very iterative and requires the ability to drill down on the `WAIT_EVENTS` systematically to eliminate serialization points.

Top Ten Mistakes Found in Oracle Systems

This section lists the most common mistakes found in Oracle systems. By following the Oracle performance improvement methodology, you should be able to avoid these mistakes altogether. If you find these mistakes in your system, then re-engineer the application where the performance effort is worthwhile. See "Automatic Performance Tuning Features" on page 1-5 for information on the features that help diagnose and tune Oracle systems. See Chapter 10, "Instance Tuning Using Performance Views" for a discussion on how wait event data reveals symptoms of problems that can be impacting performance.

1. Bad Connection Management

   The application connects and disconnects for each database interaction. This problem is common with stateless middleware in application servers. It has over two orders of magnitude impact on performance, and is totally unscalable.

2. Bad Use of Cursors and the Shared Pool

   Not using cursors results in repeated parses. If bind variables are not used, then there is hard parsing of all SQL statements. This has an order of magnitude impact in performance, and it is totally unscalable. Use cursors with bind variables that

open the cursor and execute it many times. Be suspicious of applications generating dynamic SQL.

3. Bad SQL

   Bad SQL is SQL that uses more resources than appropriate for the application requirement. This can be a decision support systems (DSS) query that runs for more than 24 hours or a query from an online application that takes more than a minute. SQL that consumes significant system resources should be investigated for potential improvement. ADDM identifies high load SQL and the SQL tuning advisor can be used to provide recommendations for improvement. See Chapter 6, "Automatic Performance Diagnostics" and Chapter 17, "Automatic SQL Tuning".

4. Use of Nonstandard Initialization Parameters

   These might have been implemented based on poor advice or incorrect assumptions. Most systems will give acceptable performance using only the set of basic parameters. In particular, parameters associated with `SPIN_COUNT` on latches and undocumented optimizer features can cause a great deal of problems that can require considerable investigation.

   Likewise, optimizer parameters set in the initialization parameter file can override proven optimal execution plans. For these reasons, schemas, schema statistics, and optimizer settings should be managed together as a group to ensure consistency of performance.

   See Also:

   - *Oracle Database Administrator's Guide* for information on initialization parameters and database creation
   - *Oracle Database Reference* for details on initialization parameters
   - "Performance Considerations for Initial Instance Configuration" on page 4-1 for information on parameters and settings in an initial instance configuration

5. Getting Database I/O Wrong

   Many sites lay out their databases poorly over the available disks. Other sites specify the number of disks incorrectly, because they configure disks by disk space and not I/O bandwidth. See Chapter 8, "I/O Configuration and Design".

6. Redo Log Setup Problems

   Many sites run with too few redo logs that are too small. Small redo logs cause system checkpoints to continuously put a high load on the buffer cache and I/O system. If there are too few redo logs, then the archive cannot keep up, and the database will wait for the archive process to catch up. See Chapter 4, "Configuring a Database for Performance" for information on sizing redo logs for performance.

7. Serialization of data blocks in the buffer cache due to lack of free lists, free list groups, transaction slots (`INITRANS`), or shortage of rollback segments.

   This is particularly common on `INSERT`-heavy applications, in applications that have raised the block size above 8K, or in applications with large numbers of active users and few rollback segments. Use automatic segment-space management (ASSM) and automatic undo management to solve this problem.

8. Long Full Table Scans

Long full table scans for high-volume or interactive online operations could indicate poor transaction design, missing indexes, or poor SQL optimization. Long table scans, by nature, are I/O intensive and unscalable.

9. High Amounts of Recursive (`SYS`) SQL

   Large amounts of recursive SQL executed by `SYS` could indicate space management activities, such as extent allocations, taking place. This is unscalable and impacts user response time. Use locally managed tablespaces to reduce recursive SQL due to extent allocation. Recursive SQL executed under another user Id is probably SQL and PL/SQL, and this is not a problem.

10. Deployment and Migration Errors

    In many cases, an application uses too many resources because the schema owning the tables has not been successfully migrated from the development environment or from an older implementation. Examples of this are missing indexes or incorrect statistics. These errors can lead to sub-optimal execution plans and poor interactive user performance. When migrating applications of known performance, export the schema statistics to maintain plan stability using the `DBMS_STATS` package.

    Although these errors are not directly detected by ADDM, ADDM highlights the resulting high load SQL.

Emergency Performance Methods

This section provides techniques for dealing with performance emergencies. You have already had the opportunity to read about a detailed methodology for establishing and improving application performance. However, in an emergency situation, a component of the system has changed to transform it from a reliable, predictable system to one that is unpredictable and not satisfying user requests.

In this case, the role of the performance engineer is to rapidly determine what has changed and take appropriate actions to resume normal service as quickly as possible. In many cases, it is necessary to take immediate action, and a rigorous performance improvement project is unrealistic.

After addressing the immediate performance problem, the performance engineer must collect sufficient debugging information either to get better clarity on the performance problem or to at least ensure that it does not happen again.

The method for debugging emergency performance problems is the same as the method described in the performance improvement method earlier in this book. However, shortcuts are taken in various stages because of the timely nature of the problem. Keeping detailed notes and records of facts found as the debugging process progresses is essential for later analysis and justification of any remedial actions. This is analogous to a doctor keeping good patient notes for future reference.

Steps in the Emergency Performance Method

The Emergency Performance Method is as follows:

1. Survey the performance problem and collect the symptoms of the performance problem. This process should include the following:

   - User feedback on how the system is underperforming. Is the problem throughput or response time?

- Ask the question, "What has changed since we last had good performance?" This answer can give clues to the problem. However, getting unbiased answers in an escalated situation can be difficult. Try to locate some reference points, such as collected statistics or log files, that were taken before and after the problem.
- Use automatic tuning features to diagnose and monitor the problem. See "Automatic Performance Tuning Features" on page 1-5 for information on the features that help diagnose and tune Oracle systems. In addition, you can use Oracle Enterprise Manager performance features to identify top SQL and sessions.

2. Sanity-check the hardware utilization of all components of the application system. Check where the highest CPU utilization is, and check the disk, memory usage, and network performance on all the system components. This quick process identifies which tier is causing the problem. If the problem is in the application, then shift analysis to application debugging. Otherwise, move on to database server analysis.
3. Determine if the database server is constrained on CPU or if it is spending time waiting on wait events. If the database server is CPU-constrained, then investigate the following:
   - Sessions that are consuming large amounts of CPU at the operating system level and database; check `V$SESS_TIME_MODEL` for database CPU usage
   - Sessions or statements that perform many buffer gets at the database level; check `V$SESSTAT` and `V$SQLSTATS`
   - Execution plan changes causing sub-optimal SQL execution; these can be difficult to locate
   - Incorrect setting of initialization parameters
   - Algorithmic issues as a result of code changes or upgrades of all components

   If the database sessions are waiting on events, then follow the wait events listed in `V$SESSION_WAIT` to determine what is causing serialization. The `V$ACTIVE_SESSION_HISTORY` view contains a sampled history of session activity which can be used to perform diagnosis even after an incident has ended and the system has returned to normal operation. In cases of massive contention for the library cache, it might not be possible to logon or submit SQL to the database. In this case, use historical data to determine why there is suddenly contention on this latch. If most waits are for I/O, then examine `V$ACTIVE_SESSION_HISTORY` to determine the SQL being run by the sessions that are performing all of the inputs and outputs. See Chapter 10, "Instance Tuning Using Performance Views" for a discussion on wait events.
4. Apply emergency action to stabilize the system. This could involve actions that take parts of the application off-line or restrict the workload that can be applied to the system. It could also involve a system restart or the termination of job in process. These naturally have service level implications.
5. Validate that the system is stable. Having made changes and restrictions to the system, validate that the system is now stable, and collect a reference set of statistics for the database. Now follow the rigorous performance method described earlier in this book to bring back all functionality and users to the system. This process may require significant application re-engineering before it is complete.

Part III

Optimizing Instance Performance

Part III describes how to tune various elements of your database system to optimize performance of an Oracle instance.

The chapters in this part are:

- Chapter 4, "Configuring a Database for Performance"
- Chapter 5, "Automatic Performance Statistics"
- Chapter 6, "Automatic Performance Diagnostics"
- Chapter 7, "Memory Configuration and Use"
- Chapter 8, "I/O Configuration and Design"
- Chapter 9, "Understanding Operating System Resources"
- Chapter 10, "Instance Tuning Using Performance Views"

4

Configuring a Database for Performance

This chapter contains an overview of the Oracle methodology for configuring a database for performance. Although performance modifications can be made to Oracle Database on an ongoing, significant benefits can be gained by proper initial configuration of the database.

This chapter contains the following sections:

Performance Considerations for Initial Instance Configuration

This section discusses some initial database instance configuration options that have important performance impacts.

If you use the Database Configuration Assistant (DBCA) to create a database, the supplied seed database includes the necessary basic initialization parameters and meets the performance recommendations that are discussed in this chapter.

See Also:

- *Oracle Database 2 Day DBA* for information creating a database with the Database Configuration Assistant
- *Oracle Database Administrator's Guide* for information about the process of creating a database

Initialization Parameters

A running Oracle instance is configured using initialization 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 few parameters shown in Table 4–2.

Table 4–1 describes the parameters necessary in a minimal initialization file. Although these parameters are necessary, they have no performance impact.

Table 4–1 Necessary Initialization Parameters Without Performance Impact

Parameter	Description
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) for each session. The setting is application-dependent; 500 is recommended.
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.

See Also: *Oracle Database Administrator's Guide* for information about managing the initialization parameters

Table 4–2 includes the most important parameters to set with performance implications:

Table 4–2 Important Initialization Parameters With Performance Impact

Parameter	Description
COMPATIBLE	Specifies the release with which the Oracle server must maintain compatibility. It lets you 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.
DB_BLOCK_SIZE	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 8192 for transaction processing systems and higher values for database warehouse systems.
SGA_TARGET	Specifies the total size of all SGA components. If SGA_TARGET is specified, then the buffer cache (DB_CACHE_SIZE), Java pool (JAVA_POOL_SIZE), large pool (LARGE_POOL_SIZE), and shared pool (SHARED_POOL_SIZE) memory pools are automatically sized. See "Automatic Shared Memory Management" on page 7-2.
PGA_AGGREGATE_TARGET	Specifies the target aggregate PGA memory available to all server processes attached to the instance. See "PGA Memory Management" on page 7-42 for information on PGA memory management.
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.

Table 4–2 (Cont.) Important Initialization Parameters With Performance Impact

Parameter	Description
UNDO_MANAGEMENT	Specifies the undo space management mode used by Oracle Database. The default value is AUTO. If unspecified, AUTO will be used.
UNDO_TABLESPACE	Specifies the undo tablespace to be used when an instance starts up.

See Also:

- Chapter 7, "Memory Configuration and Use"
- *Oracle Database Reference* for information on initialization parameters
- *Oracle Streams Concepts and Administration* for information on the STREAMS_POOL_SIZE initialization parameter

Configuring Undo Space

Oracle Database uses undo space to store data used for read consistency, recovery purposes, and actual rollback statements. This data is saved in one or more undo tablespaces. If you use the Database Configuration Assistant (DBCA) to create a database, the undo tablespace will be created automatically. To manually create an undo tablespace, add the UNDO TABLESPACE clause in the CREATE DATABASE statement when creating the database.

To automate the management of undo data, Oracle Database uses automatic undo management, which transparently creates and manages undo segments. Automatic undo management is controlled by the UNDO_MANAGEMENT initialization parameter. To enable automatic undo management, set the UNDO_MANAGEMENT initialization parameter to AUTO (the default setting). If unspecified, the UNDO_MANAGEMENT initialization parameter will use the AUTO setting. Oracle strongly recommends using automatic undo management, because it significantly simplifies database management and eliminates the need for any manual tuning of undo (rollback) segments. Manual undo management using rollback segments is supported for backward compatibility reasons.

The V$UNDOSTAT view contains statistics for monitoring and tuning undo space. Using this view, you can better estimate the amount of undo space required for the current workload. Oracle Database also uses this information to help tune undo usage. The V$ROLLSTAT view contains information about the behavior of the undo segments in the undo tablespace.

See Also:

- *Oracle Database 2 Day DBA* and Oracle Enterprise Manager online help for information about the Undo Management Advisor
- *Oracle Database Administrator's Guide* for information on managing undo space using automatic undo management
- *Oracle Database Reference* for information about the dynamic performance V$ROLLSTAT and V$UNDOSTAT views

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. Undersized log files increase checkpoint activity and reduce performance.

Although the size of the redo log files does not affect LGWR performance, it can affect DBWR and checkpoint behavior. Checkpoint frequency is affected by several factors, including log file size and the setting of the `FAST_START_MTTR_TARGET` initialization parameter. If the `FAST_START_MTTR_TARGET` parameter is set to limit the instance recovery time, Oracle automatically tries to checkpoint as frequently as necessary. Under this condition, the size of the log files should be large enough to avoid additional checkpointing due to under sized log files. The optimal size can be obtained by querying the `OPTIMAL_LOGFILE_SIZE` column from the `V$INSTANCE_RECOVERY` view. You can also obtain sizing advice on the **Redo Log Groups** page of Oracle Enterprise Manager Database Control.

It may not always be 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.

> **See Also:** *Oracle Database Administrator's Guide* for information on managing the redo log

Creating Subsequent Tablespaces

If you use the Database Configuration Assistant (DBCA) to create a database, the supplied seed database automatically includes all the necessary tablespaces. If you choose not to use DBCA, you need to create extra tablespaces after creating the initial database.

All databases should have several tablespaces in addition to the SYSTEM and SYSAUX tablespaces. These additional tablespaces include:

- A temporary tablespace, which is used for things like sorting
- An undo tablespace to contain information for read consistency, recovery, and rollback statements
- 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 special empty block images. Temporary files are not initialized.

Oracle does this to ensure that all datafiles can be written in their entirety, but this can obviously be a lengthy process if done serially. Therefore, run multiple `CREATE TABLESPACE` statements concurrently to speed up the tablespace creation process. 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.

Creating Permanent Tablespaces - Automatic Segment-Space Management

For permanent tablespaces, Oracle recommends using automatic segment-space management. Such tablespaces, often referred to as bitmap tablespaces, are locally managed tablespaces with bitmap segment space management.

See Also:

- *Oracle Database Concepts* for a discussion of free space management
- *Oracle Database Administrator's Guide* for more information on creating and using automatic segment-space management for tablespaces

Creating Temporary Tablespaces

Properly configuring the temporary tablespace helps optimize disk sort performance. Temporary tablespaces can be dictionary-managed or locally managed. Oracle recommends the use of locally managed temporary tablespaces with a `UNIFORM` extent size of 1 `MB`.

You should monitor temporary tablespace activity to check how many extents are being allocated for the temporary segment. If an application extensively uses temporary tables, as in a situation when many users are concurrently using temporary tables, the extent size could be set smaller, such as 256K, because every usage requires at least one extent. The `EXTENT MANAGEMENT LOCAL` clause is optional for temporary tablespaces because all temporary tablespaces are created with locally managed extents of a uniform size. The default for `SIZE` is `1M`.

See Also:

- *Oracle Database Administrator's Guide* for more information on managing temporary tablespaces
- *Oracle Database Concepts* for more information on temporary tablespaces
- *Oracle Database SQL Reference* for more information on using the `CREATE` and `ALTER TABLESPACE` statements with the `TEMPORARY` clause

Creating and Maintaining Tables for Optimal Performance

When installing applications, an initial step is to create all necessary tables and indexes. 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 creating tables and indexes, note the following:

- Specify automatic segment-space management for tablespaces

  This allows Oracle to automatically manage segment space for best performance.

- Set storage options carefully

  Applications should carefully set storage options for the intended use of the table or index. This includes setting the value for `PCTFREE`. Note that using automatic segment-space management eliminates the need to specify `PCTUSED`.

Note: Use of free lists is no longer encouraged. To use automatic segment-space management, create locally managed tablespaces, with the segment space management clause set to `AUTO`.

Table Compression

Heap-organized tables can be stored in a compressed format that is transparent for any kind of application. Table compression was designed primarily for read-only environments and can cause processing overhead for DML operations in some cases. However, it increases performance for many read operations, especially when your system is I/O bound.

Compressed data in a database block is self-contained which means that all the information needed to re-create the uncompressed data in a block is available within that block. A block will also be kept compressed in the buffer cache. Table compression not only reduces the disk storage but also the memory usage, specifically the buffer cache requirements. Performance improvements are accomplished by reducing the amount of necessary I/O operations for accessing a table and by increasing the probability of buffer cache hits.

Estimating the Compression factor

Table compression works by eliminating column value repetitions within individual blocks. Duplicate values in all the rows and columns in a block are stored once at the beginning of the block, in what is called a symbol table for that block. All occurrences of such values are replaced with a short reference to the symbol table. The compression is higher in blocks that have more repeated values.

Before compressing large tables you should estimate the expected compression factor. The compression factor is defined as the number of blocks necessary to store the information in an uncompressed form divided by the number of blocks necessary for a compressed storage. The compression factor can be estimated by sampling a small number of representative data blocks of the table to be compressed and comparing the average number of records for each block for the uncompressed and compressed case. Experience shows that approximately 1000 data blocks provides a very accurate estimation of the compression factor. Note that the more blocks you are sampling, the more accurate the result become.

Tuning to Achieve a Better Compression Ratio

Oracle achieves a good compression factor in many cases with no special tuning. As a database administrator or application developer, you can try to tune the compression factor by reorganizing the records when the compression actually takes place. Tuning can improve the compression factor slightly in some cases and very substantially in other cases.

To improve the compression factor you have to increase the likelihood of value repetitions within a database block. The compression factor that can be achieved depends on the cardinality of a specific column or column pairs (representing the likelihood of column value repetitions) and on the average row length of those columns. Oracle table compression not only compresses duplicate values of a single column but tries to use multi-column value pairs whenever possible. Without a very detailed understanding of the data distribution it is very difficult to predict the most optimal order.

See Also: *Oracle Database Data Warehousing Guide* for information on table compression and partitions

Reclaiming Unused Space

Over time, it is common for segment space to become fragmented or for a segment to acquire a lot of free space as the result of update and delete operations. The resulting sparsely populated objects can suffer performance degradation during queries and DML operations.

Oracle Database provides a Segment Advisor that provides advice on whether an object has space available for reclamation based on the level of space fragmentation within an object.

> **See Also:** *Oracle Database Administrator's Guide* and *Oracle Database 2 Day DBA* for information about the Segment Advisor

If an object does have space available for reclamation, you can compact and shrink database segments or you can deallocate unused space at the end of a database segment.

> **See Also:**
>
> - *Oracle Database Administrator's Guide* for a discussion of reclaiming unused space
> - *Oracle Database SQL Reference* for details about the SQL statements used to shrink database segments or deallocate unused space

Indexing Data

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.

> **See Also:** *Oracle Database Utilities* for information on SQL*Loader

Specifying Memory for Sorting 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. Oracle recommends that you enable automatic sizing of SQL working areas by setting the `PGA_AGGREGATE_TARGET` initialization parameter.

> **See Also:**
>
> - "PGA Memory Management" on page 7-42 for information on PGA memory management
> - *Oracle Database Reference* for information on the `PGA_AGGREGATE_TARGET` initialization parameter

Performance Considerations for Shared Servers

Using shared servers reduces the number of processes and the amount of memory consumed on the server machine. Shared servers are beneficial for systems where there are many OLTP users performing intermittent transactions.

Using shared servers rather than dedicated servers is also generally better for systems that have a high connection rate to the database. With shared servers, when a connect request is received, a dispatcher is already available to handle concurrent connection requests. With dedicated servers, on the other hand, a connection-specific dedicated server is sequentially initialized for each connection request.

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, a session can be prevented from migrating to another shared server while parallel execution is active.

A session can remain nonmigratable even after a request from the client has been processed, because not all the user information has been stored in the UGA. If a server were to process the request from the client, then the part of the user state that was not stored in the UGA would be inaccessible. To avoid this, individual shared servers often need to remain bound to a user session.

See Also:

- *Oracle Database Administrator's Guide* for information on managing shared servers
- *Oracle Database Net Services Administrator's Guide* for information on configuring dispatchers for shared servers

When using some features, you may need to configure more shared servers, 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 architecture:

- Identifying Contention Using the Dispatcher-Specific Views
- Identifying Contention for Shared Servers

Identifying Contention Using the Dispatcher-Specific Views

The following views provide dispatcher performance statistics:

- `V$DISPATCHER` - general information about dispatcher processes
- `V$DISPATCHER_RATE` - dispatcher processing 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 sample. 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 less than the maximum, then you likely have an optimally tuned shared server environment.

If the current and average rates are significantly less than 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.

See Also:

- *Oracle Database Reference* for detailed information about the V$DISPATCHER and V$DISPATCHER_RATE views
- *Oracle Enterprise Manager Concepts* for information about Oracle Tuning Pack applications that monitor statistics

Reducing Contention for Dispatcher Processes

To reduce contention, consider the following:

- Adding dispatcher processes

  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.

- 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.

- Enabling Session Multiplexing

  Multiplexing is used by a connection manager process to establish and maintain network sessions 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. Session multiplexing is beneficial because it maximizes use of the dispatcher process connections. Multiplexing is also useful for multiplexing database link sessions between dispatchers.

See Also:

- *Oracle Database Administrator's Guide* for information on configuring dispatcher processes
- *Oracle Database Net Services Administrator's Guide* for information on configuring connection pooling
- *Oracle Database Reference* for information about the DISPATCHERS and MAX_DISPATCHERS parameters

Identifying Contention for Shared Servers

This section discusses how to identify 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. Table 4–3 lists the columns showing the wait times for requests and the number of requests in the queue.

Table 4–3 Wait Time and Request Columns in V$QUEUE

Column	Description
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. You can do this by modifying the optional server process initialization parameters:

- MAX_DISPATCHERS
- MAX_SHARED_SERVERS
- DISPATCHERS
- SHARED_SERVERS

> **See Also:** *Oracle Database Administrator's Guide* for information on setting the shared server process initialization parameters

5

Automatic Performance Statistics

This chapter discusses the gathering of performance statistics. This chapter contains the following topics:

- Overview of Data Gathering
- Overview of the Automatic Workload Repository
- Managing the Automatic Workload Repository

Overview of Data Gathering

To effectively diagnose performance problems, statistics must be available. Oracle generates many types of cumulative statistics for the system, sessions, and individual SQL statements. Oracle also tracks cumulative statistics on segments and services. When analyzing a performance problem in any of these scopes, you typically look at the change in statistics (delta value) over the period of time you are interested in. Specifically, you look at the difference between the cumulative value of a statistic at the start of the period and the cumulative value at the end.

Cumulative values for statistics are generally available through dynamic performance views, such as the `V$SESSTAT` and `V$SYSSTAT` views. Note that the cumulative values in dynamic views are reset when the database instance is shutdown. The Automatic Workload Repository (AWR) automatically persists the cumulative and delta values for most of the statistics at all levels except the session level. This process is repeated on a regular time period and the result is called an AWR snapshot. The delta values captured by the snapshot represent the changes for each statistic over the time period. See "Overview of the Automatic Workload Repository" on page 5-8.

Another type of statistic collected by Oracle is called a metric. A metric is defined as the rate of change in some cumulative statistic. That rate can be measured against a variety of units, including time, transactions, or database calls. For example, the number database calls per second is a metric. Metric values are exposed in some V$ views, where the values are the average over a fairly small time interval, typically 60 seconds. A history of recent metric values is available through V$ views, and some of the data is also persisted by AWR snapshots.

A third type of statistical data collected by Oracle is sampled data. This sampling is performed by the active session history (ASH) sampler. ASH samples the current state of all active sessions. This data is collected into memory and can be accessed by a V$ view. It is also written out to persistent store by the AWR snapshot processing. See "Active Session History" on page 5-3.

A powerful tool for diagnosing performance problems is the use of statistical baselines. A statistical baseline is collection of statistic rates usually taken over time period where the system is performing well at peak load. Comparing statistics

captured during a period of bad performance to a baseline helps discover specific statistics that have increased significantly and could be the cause of the problem.

AWR supports the capture of baseline data by enabling you to specify and preserve a pair or range of AWR snapshots as a baseline. Carefully consider the time period you choose as a baseline; the baseline should be a good representation of the peak load on the system. In the future, you can compare these baselines with snapshots captured during periods of poor performance.

Oracle Enterprise Manager is the recommended tool for viewing both real time data in the dynamic performance views and historical data from the AWR history tables. Enterprise Manager can also be used to capture operating system and network statistical data that can be correlated with AWR data. For more information, see *Oracle Database 2 Day + Performance Tuning Guide*.

This section covers the following topics:

- Database Statistics
- Operating System Statistics
- Interpreting Statistics

Database Statistics

Database statistics provide information on the type of load on the database, as well as the internal and external resources used by the database. This section describes some of the more important statistics:

- Wait Events
- Time Model Statistics
- Active Session History
- System and Session 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.

To enable easier high-level analysis of the wait events, the events are grouped into classes. The wait event classes include: Administrative, Application, Cluster, Commit, Concurrency, Configuration, Idle, Network, Other, Scheduler, System I/O, and User I/O.

The wait classes are based on a common solution that usually applies to fixing a problem with the wait event. For example, exclusive TX locks are generally an application level issue and HW locks are generally a configuration issue.

The following list includes common examples of the waits in some of the classes:

- Application: locks waits caused by row level locking or explicit lock commands
- Commit: waits for redo log write confirmation after a commit
- Idle: wait events that signify the session is inactive, such as `SQL*Net message from client`
- Network: waits for data to be sent over the network
- User I/O: wait for blocks to be read off a disk

Wait event statistics for an instance include statistics for both background and foreground processes. Because you would typically focus your effort in tuning foreground activities, overall instance activity is broken down into foreground and background statistics in the relevant V$ views to facilitate tuning.

The V$SYSTEM_EVENT view provides wait event statistics for the foreground activities of an instance as well as the wait event statistics for the instance as a whole. The V$SYSTEM_WAIT_CLASS view provides these wait event statistics (foreground and instance statistics) after aggregating to wait classes.

The V$SESSION_EVENT and V$SYSTEM_WAIT_CLASS views provide wait event and wait class statistics at session level.

> **See Also:** *Oracle Database Reference* for more information about Oracle wait events

Time Model Statistics

When tuning an Oracle system, each component has its own set of statistics. To look at the system as a whole, it is necessary to have a common scale for comparisons. Because of this, most Oracle advisories and reports describe statistics in terms of time. In addition, the V$SESS_TIME_MODEL and V$SYS_TIME_MODEL views provide time model statistics. Using the common time instrumentation helps to identify quantitative effects on the database operations.

The most important of the time model statistics is DB time. This statistics represents the total time spent in database calls and is an indicator of the total instance workload. It is calculated by aggregating the CPU and wait times of all sessions not waiting on idle wait events (non-idle user sessions).

DB time is measured cumulatively from the time that the instance was started. Because DB time it is calculated by combining the times from all non-idle user sessions, it is possible that the DB time can exceed the actual time elapsed since the instance started up. For example, an instance that has been running for 30 minutes could have four active user sessions whose cumulative DB time is approximately 120 minutes.

The objective for tuning an Oracle system could be stated as reducing the time that users spend in performing some action on the database, or simply reducing DB time. Other time model statistics provide quantitative effects (in time) on specific actions, such as logon operations and hard and soft parses.

> **See Also:** *Oracle Database Reference* for information about the V$SESS_TIME_MODEL and V$SYS_TIME_MODEL views

Active Session History

The V$ACTIVE_SESSION_HISTORY view provides sampled session activity in the instance. Active sessions are sampled every second and are stored in a circular buffer in SGA. Any session that is connected to the database and is waiting for an event that does not belong to the Idle wait class is considered as an active session. This includes any session that was on the CPU at the time of sampling.

Each session sample is a set of rows and the V$ACTIVE_SESSION_HISTORY view returns one row for each active session per sample, returning the latest session sample rows first. Because the active session samples are stored in a circular buffer in SGA, the greater the system activity, the smaller the number of seconds of session activity that can be stored in the circular buffer. This means that the duration for which a session sample appears in the V$ view, or the number of seconds of session activity that is displayed in the V$ view, is completely dependent on the database activity.

As part of the Automatic Workload Repository (AWR) snapshots, the content of V$ACTIVE_SESSION_HISTORY is also flushed to disk. Because the content of this V$ view can get quite large during heavy system activity, only a portion of the session samples is written to disk.

By capturing only active sessions, a manageable set of data is represented with the size being directly related to the work being performed rather than the number of sessions allowed on the system. Using the Active Session History enables you to examine and perform detailed analysis on both current data in the V$ACTIVE_SESSION_HISTORY view and historical data in the DBA_HIST_ACTIVE_SESS_HISTORY view, often avoiding the need to replay the workload to gather additional performance tracing information. Active Session History also contains execution plan information for each SQL statement that is captured. This information can be used to identify which part of the SQL execution contributed most significantly to the SQL elapsed time. The data present in ASH can be rolled up on various dimensions that it captures, including the following:

- SQL identifier of SQL statement
- SQL plan identifier and hash value of the SQL plan used to execute the SQL statement
- SQL execution plan information
- Object number, file number, and block number
- Wait event identifier and parameters
- Session identifier and session serial number
- Module and action name
- Client identifier of the session
- Service hash identifier
- Consumer group identifier

> **See Also:** *Oracle Database Reference* for more information about the V$ACTIVE_SESSION_HISTORY view

Active Session History information over a specified duration can be gathered into a report. For more information, see "Generating Active Session History Reports" on page 5-28.

System and Session Statistics

A large number of cumulative database statistics are available on a system and session level through the V$SYSSTAT and V$SESSTAT views.

> **See Also:** *Oracle Database Reference* for information about the V$SYSSTAT and V$SESSTAT views

Operating System Statistics

Operating system statistics provide information on the usage and performance of the main hardware components of the system, as well as the performance of the operating system itself. This information is crucial for detecting potential resource exhaustion, such as CPU cycles and physical memory, and for detecting bad performance of peripherals, such as disk drives.

Operating system statistics are only an indication of how the hardware and operating system are working. Many system performance analysts react to a hardware resource shortage by installing more hardware. This is a reactionary response to a series of symptoms shown in the operating system statistics. It is always best to consider operating system statistics as a diagnostic tool, similar to the way many doctors use body temperature, pulse rate, and patient pain when making a diagnosis. To help identify bottlenecks, gather operating system statistics for all servers in the system under performance analysis.

Operating system statistics include the following:

- CPU Statistics
- Virtual Memory Statistics
- Disk I/O Statistics
- Network Statistics

For information on tools for gathering operating statistics, see "Operating System Data Gathering Tools" on page 5-6.

CPU Statistics

CPU utilization is the most important operating system statistic in the tuning process. Get CPU utilization for the entire system and for each individual CPU on multi-processor environments. Utilization for each CPU can detect single-threading and scalability issues.

Most operating systems report CPU usage as time spent in user space or mode and time spent in kernel space or mode. These additional statistics allow better analysis of what is actually being executed on the CPU.

On an Oracle data server system, where there is generally only one application running, the server runs database activity in user space. Activities required to service database requests (such as scheduling, synchronization, I/O, memory management, and process/thread creation and tear down) run in kernel mode. In a system where all CPU is fully utilized, a healthy Oracle system runs between 65% and 95% in user space.

The `V$OSSTAT` view captures machine level information in the database, making it easier for you to determine if there are hardware level resource issues. The `V$SYSMETRIC_HISTORY` view shows a one-hour history of the Host CPU Utilization metric, a representation of percentage of CPU usage at each one-minute interval. The `V$SYS_TIME_MODEL` view supplies statistics on the CPU usage by the Oracle database. Using both sets of statistics enable you to determine whether the Oracle database or other system activity is the cause of the CPU problems.

Virtual Memory Statistics

Virtual memory statistics should mainly be used as a check to validate that there is very little paging or swapping activity on the system. System performance degrades rapidly and unpredictably when paging or swapping occurs.

Individual process memory statistics can detect memory leaks due to a programming failure to deallocate memory taken from the process heap. These statistics should be used to validate that memory usage does not increase after the system has reached a steady state after startup. This problem is particularly acute on shared server applications on middle tier machines where session state may persist across user interactions, and on completion state information that is not fully deallocated.

Disk I/O Statistics

Because the database resides on a set of disks, the performance of the I/O subsystem is very important to the performance of the database. Most operating systems provide extensive statistics on disk performance. The most important disk statistics are the current response time and the length of the disk queues. These statistics show if the disk is performing optimally or if the disk is being overworked.

Measure the normal performance of the I/O system; typical values for a single block read range from 5 to 20 milliseconds, depending on the hardware used. If the hardware shows response times much higher than the normal performance value, then it is performing badly or is overworked. This is your bottleneck. If disk queues start to exceed two, then the disk is a potential bottleneck of the system.

Oracle Database also maintains a consistent set of I/O statistics for the I/O calls it issues. These statistics are captured for both single and multi block read and write operations in the following dimensions:

- Consumer group

  When Oracle Database Resource Manager is enabled, I/O statistics for all consumer groups that are part of the currently enabled resource plan are captured in the `V$IOSTAT_CONSUMER_GROUP` view. These cumulative statistics will be sampled every hour and stored as historical statistics in the AWR.

- Database file

  I/O statistics of database files that are or have been accessed are captured in the `V$IOSTAT_FILE` view.

- Database function

  I/O statistics for database functions (such as the LGWR and DBWR) are captured in the `V$IOSTAT_FUNCTION` view

For information about using views in Oracle Database to identify I/O problems, see "Identifying I/O Problems Using V$ Views" on page 10-4.

Network Statistics

Network statistics can be used in much the same way as disk statistics to determine if a network or network interface is overloaded or not performing optimally. In today's networked applications, network latency can be a large portion of the actual user response time. For this reason, these statistics are a crucial debugging tool.

Oracle Database maintains a set of network I/O statistics in the `V$IOSTAT_NETWORK` view. For information about using the `V$IOSTAT_NETWORK` view in Oracle Database to identify network issues, see "Identifying Network Issues" on page 10-6.

Operating System Data Gathering Tools

Table 5–1 shows the various tools for gathering operating statistics on UNIX. For Windows, use the Performance Monitor tool.

Table 5–1 UNIX Tools for Operating Statistics

Component	UNIX Tool
CPU	sar, vmstat, mpstat, iostat
Memory	sar, vmstat
Disk	sar, iostat
Network	netstat

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. Also, because foreground activities are tunable, it is better to first analyze the statistics from foreground activities before analyzing the statistics from background activities.

Some pitfalls are discussed in the following sections:

- 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 or not there is 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. See "Calculating the Buffer Cache Hit Ratio" on page 7-9.

- 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. See "Wait Events" on page 5-2.

  Note: Timed statistics are automatically collected for the database if the initialization parameter `STATISTICS_LEVEL` is set to `TYPICAL` or `ALL`. If `STATISTICS_LEVEL` is set to `BASIC`, then you must set `TIMED_STATISTICS` to `TRUE` to enable collection of timed statistics. Note that setting `STATISTICS_LEVEL` to `BASIC` disables many automatic features and is not recommended.

  If you explicitly set `DB_CACHE_ADVICE`, `TIMED_STATISTICS`, or `TIMED_OS_STATISTICS`, either in the initialization parameter file or by using `ALTER_SYSTEM` or `ALTER SESSION`, the explicitly set value overrides the value derived from `STATISTICS_LEVEL`.

- 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. See "Idle Wait Events" on page 10-29.

- Computed statistics

  When interpreting computed statistics (such as rates, statistics normalized over transactions, 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.

See Also:

- "Setting the Level of Statistics Collection" on page 10-7 for information about `STATISTICS_LEVEL` settings
- *Oracle Database Reference* for information on the `STATISTICS_LEVEL` initialization parameter

Overview of the Automatic Workload Repository

The Automatic Workload Repository (AWR) collects, processes, and maintains performance statistics for problem detection and self-tuning purposes. This data is both in memory and stored in the database. The gathered data can be displayed in both reports and views.

The statistics collected and processed by AWR include:

- Object statistics that determine both access and usage statistics of database segments
- Time model statistics based on time usage for activities, displayed in the `V$SYS_TIME_MODEL` and `V$SESS_TIME_MODEL` views
- Some of the system and session statistics collected in the `V$SYSSTAT` and `V$SESSTAT` views
- SQL statements that are producing the highest load on the system, based on criteria such as elapsed time and CPU time
- Active Session History (ASH) statistics, representing the history of recent sessions activity

Gathering database statistics using the AWR is enabled by default and is controlled by the `STATISTICS_LEVEL` initialization parameter. The `STATISTICS_LEVEL` initialization parameter should be set to the `TYPICAL` or `ALL` to enable statistics gathering by the AWR. The default setting is `TYPICAL`. Setting the `STATISTICS_LEVEL` parameter to `BASIC` disables many Oracle Database features,

including the AWR, and is not recommended. If the STATISTICS_LEVEL parameter is set to BASIC, you can still manually capture AWR statistics using the DBMS_WORKLOAD_REPOSITORY package. However, because in-memory collection of many system statistics—such as segments statistics and memory advisor information—will be disabled, the statistics captured in these snapshots may not be complete. For information about the STATISTICS_LEVEL initialization parameter, see *Oracle Database Reference*.

This section describes the Automatic Workload Repository and contains the following topics:

- Snapshots
- Baselines
- Space Consumption

Snapshots

Snapshots are sets of historical data for specific time periods that are used for performance comparisons by ADDM. By default, Oracle Database automatically generates snapshots of the performance data once every hour and retains the statistics in the workload repository for 8 days. You can also manually create snapshots, but this is usually not necessary. The data in the snapshot interval is then analyzed by the Automatic Database Diagnostic Monitor (ADDM). For information about ADDM, see "Overview of the Automatic Database Diagnostic Monitor" on page 6-1.

AWR compares the difference between snapshots to determine which SQL statements to capture based on the effect on the system load. This reduces the number of SQL statements that need to be captured over time.

For information about managing snapshots, see "Managing Snapshots" on page 5-12.

Baselines

A baseline contains performance data from a specific time period that is preserved for comparison with other similar workload periods when performance problems occur. The snapshots contained in a baseline are excluded from the automatic AWR purging process and are retained indefinitely.

There are several types of available baselines in Oracle Database:

- Fixed Baselines
- Moving Window Baseline
- Baseline Templates

Fixed Baselines

A fixed baseline corresponds to a fixed, contiguous time period in the past that you specify. Before creating a fixed baseline, carefully consider the time period you choose as a baseline, because the baseline should represent the system operating at an optimal level. In the future, you can compare the baseline with other baselines or snapshots captured during periods of poor performance to analyze performance degradation over time.

For information about managing fixed baselines, see "Managing Baselines" on page 5-13.

Moving Window Baseline

A moving window baseline corresponds to all AWR data that exists within the AWR retention period. This is useful when using adaptive thresholds because the AWR data in the entire AWR retention period can be used to compute metric threshold values.

Oracle Database automatically maintains a system-defined moving window baseline. The default window size for the system-defined moving window baseline is the current AWR retention period, which by default is 8 days. If you are planning to use adaptive thresholds, consider using a larger moving window—such as 30 days—to accurately compute threshold values. You can resize the moving window baseline by changing the number of days in the moving window to a value that is equal to or less than the number of days in the AWR retention period. Therefore, to increase the size of a moving window, you will first need to increase the AWR retention period accordingly.

For information about resizing the moving window baseline, see "Modifying the Window Size of the Default Moving Window Baseline" on page 5-15.

Baseline Templates

You can also create baselines for a contiguous time period in the future using baseline templates. There are two types of baseline templates: single and repeating.

A single baseline template can be used to create a baseline for a single contiguous time period in the future. This is useful if you know beforehand of a time period that you want to capture in the future. For example, you may want to capture the AWR data during a system test that is scheduled for the upcoming weekend. In this case, you can create a single baseline template to automatically capture the time period when the test will take place.

A repeating baseline template can be used to create and drop baselines based on a repeating time schedule. This is useful if you want Oracle Database to automatically capture a contiguous time period on an ongoing basis. For example, you may want to capture the AWR data during every Monday morning for a month. In this case, you can create a repeating baseline template to automatically create baselines on a repeating schedule for every Monday, and automatically remove older baselines after a specified expiration interval such as one month.

For information about managing baseline templates, see "Managing Baseline Templates" on page 5-16.

Space Consumption

The space consumed by the Automatic Workload Repository is determined by several factors:

- Number of active sessions in the system at any given time
- Snapshot interval

  The snapshot interval determines the frequency at which snapshots are captured. A smaller snapshot interval increases the frequency, which increases the volume of data collected by the Automatic Workload Repository.
- Historical data retention period

  The retention period determines how long this data is retained before being purged. A longer retention period increases the space consumed by the Automatic Workload Repository.

By default, snapshots are captured once every hour and are retained in the database for 8 days. With these default settings, a typical system with an average of 10 concurrent active sessions can require approximately 200 to 300 MB of space for its AWR data. It is possible to change the default values for both snapshot interval and retention period. For more information, see "Modifying Snapshot Settings" on page 5-13 for information on modifying AWR settings.

The Automatic Workload Repository space consumption can be reduced by the increasing the snapshot interval and reducing the retention period. When reducing the retention period, note that several Oracle self-managing features depend on AWR data for proper functioning. Not having enough data can affect the validity and accuracy of these components and features, including:

- Automatic Database Diagnostic Monitor
- SQL Tuning Advisor
- Undo Advisor
- Segment Advisor

If possible, Oracle recommends that you set the AWR retention period large enough to capture at least one complete workload cycle. If your system experiences weekly workload cycles, such as OLTP workload during weekdays and batch jobs during the weekend, you do not need to change the default AWR retention period of 8 days. However if your system is subjected to a monthly peak load during month end book closing, you may have to set the retention period to one month.

Under exceptional circumstances, the automatic snapshot collection can be completely turned off by setting the snapshot interval to 0. Under this condition, the automatic collection of the workload and statistical data is stopped and much of the Oracle self-management functionality is not operational. In addition, you will not be able to manually create snapshots. For this reason, Oracle strongly recommends that you do not turn off the automatic snapshot collection.

Managing the Automatic Workload Repository

This section describes how to manage the Automatic Workload Repository and contains the following topics:

- Managing Snapshots
- Managing Baselines
- Managing Baseline Templates
- Transporting Automatic Workload Repository Data
- Using Automatic Workload Repository Views
- Generating Automatic Workload Repository Reports
- Generating Automatic Workload Repository Compare Periods Reports
- Generating Active Session History Reports
- Using Active Session History Reports

For a description of the Automatic Workload Repository, see "Overview of the Automatic Workload Repository" on page 5-8.

Managing Snapshots

By default, Oracle Database generates snapshots once every hour, and retains the statistics in the workload repository for 8 days. When necessary, you can use DBMS_WORKLOAD_REPOSITORY procedures to manually create, drop, and modify the snapshots. To invoke these procedures, a user must be granted the DBA role. For more information about snapshots, see "Snapshots" on page 5-9.

The primary interface for managing snapshots is Oracle Enterprise Manager. Whenever possible, you should manage snapshots using Oracle Enterprise Manager, as described in *Oracle Database 2 Day + Performance Tuning Guide*. If Oracle Enterprise Manager is unavailable, you can manage snapshots using the DBMS_WORKLOAD_REPOSITORY package, as described in the following sections:

- Creating Snapshots
- Dropping Snapshots
- Modifying Snapshot Settings

> **See Also:** *Oracle Database PL/SQL Packages and Types Reference* for detailed information on the DBMS_WORKLOAD_REPOSITORY package

Creating Snapshots

You can manually create snapshots with the CREATE_SNAPSHOT procedure if you want to capture statistics at times different than those of the automatically generated snapshots. For example:

```
BEGIN
  DBMS_WORKLOAD_REPOSITORY.CREATE_SNAPSHOT ();
END;
/
```

In this example, a snapshot for the instance is created immediately with the flush level specified to the default flush level of TYPICAL. You can view this snapshot in the DBA_HIST_SNAPSHOT view.

Dropping Snapshots

You can drop a range of snapshots using the DROP_SNAPSHOT_RANGE procedure. To view a list of the snapshot Ids along with database Ids, check the DBA_HIST_SNAPSHOT view. For example, you can drop the following range of snapshots:

```
BEGIN
  DBMS_WORKLOAD_REPOSITORY.DROP_SNAPSHOT_RANGE (low_snap_id => 22,
                           high_snap_id => 32, dbid => 3310949047);
END;
/
```

In the example, the range of snapshot Ids to drop is specified from 22 to 32. The optional database identifier is 3310949047. If you do not specify a value for dbid, the local database identifier is used as the default value.

Active Session History data (ASH) that belongs to the time period specified by the snapshot range is also purged when the DROP_SNAPSHOT_RANGE procedure is called.

Modifying Snapshot Settings

You can adjust the interval, retention, and captured Top SQL of snapshot generation for a specified database Id, but note that this can affect the precision of the Oracle diagnostic tools.

The `INTERVAL` setting affects how often in minutes that snapshots are automatically generated. The `RETENTION` setting affects how long in minutes that snapshots are stored in the workload repository. The `TOPNSQL` setting affects the number of Top SQL to flush for each SQL criteria (Elapsed Time, CPU Time, Parse Calls, Shareable Memory, and Version Count). The value for this setting will not be affected by the statistics/flush level and will override the system default behavior for the AWR SQL collection. It is possible to set the value for this setting to `MAXIMUM` to capture the complete set of SQL in the cursor cache, though by doing so (or by setting the value to a very high number) may lead to possible space and performance issues since there will more data to collect and store. To adjust the settings, use the `MODIFY_SNAPSHOT_SETTINGS` procedure. For example:

```
BEGIN
  DBMS_WORKLOAD_REPOSITORY.MODIFY_SNAPSHOT_SETTINGS( retention => 43200,
                interval => 30, topnsql => 100, dbid => 3310949047);
END;
/
```

In this example, the retention period is specified as 43200 minutes (30 days), the interval between each snapshot is specified as 30 minutes, and the number of Top SQL to flush for each SQL criteria as 100. If NULL is specified, the existing value is preserved. The optional database identifier is `3310949047`. If you do not specify a value for `dbid`, the local database identifier is used as the default value. You can check the current settings for your database instance with the `DBA_HIST_WR_CONTROL` view.

Managing Baselines

This section describes how to manage baselines. For more information about baselines, see "Baselines" on page 5-9.

The primary interface for managing baselines is Oracle Enterprise Manager. Whenever possible, you should manage baselines using Oracle Enterprise Manager, as described in *Oracle Database 2 Day + Performance Tuning Guide*. If Oracle Enterprise Manager is unavailable, you can manage baselines using the `DBMS_WORKLOAD_REPOSITORY` package, as described in the following sections:

This section contains the following topics:

- Creating a Baseline
- Dropping a Baseline
- Renaming a Baseline
- Displaying Baseline Metrics
- Modifying the Window Size of the Default Moving Window Baseline

> **See Also:** *Oracle Database PL/SQL Packages and Types Reference* for detailed information on the `DBMS_WORKLOAD_REPOSITORY` package

Creating a Baseline

This section describes how to create a baseline using an existing range of snapshots.

To create a baseline:

1. Review the existing snapshots in the DBA_HIST_SNAPSHOT view to determine the range of snapshots that you want to use.
2. Use the CREATE_BASELINE procedure to create a baseline using the desired range of snapshots:

```
BEGIN
    DBMS_WORKLOAD_REPOSITORY.CREATE_BASELINE (start_snap_id => 270,
                   end_snap_id => 280, baseline_name => 'peak baseline',
                   dbid => 3310949047, expiration => 30);
END;
/
```

   In this example, 270 is the start snapshot sequence number and 280 is the end snapshot sequence. The name of baseline is peak baseline. The optional database identifier is 3310949047. If you do not specify a value for dbid, the local database identifier is used as the default value. The optional expiration parameter is set to 30, so the baseline will expire and be dropped automatically after 30 days. If you do not specify a value for expiration, the baseline will never expire.

The system automatically assign a unique baseline Id to the new baseline when the baseline is created. The baseline Id and database identifier are displayed in the DBA_HIST_BASELINE view.

Dropping a Baseline

This section describes how to drop an existing baseline. Periodically, you may want to drop a baseline that is no longer used to conserve disk space. The snapshots associated with a baseline are retained indefinitely until you explicitly drop the baseline or the baseline has expired.

To drop a baseline:

1. Review the existing baselines in the DBA_HIST_BASELINE view to determine the baseline that you want to drop.
2. Use the DROP_BASELINE procedure to drop the desired baseline:

```
BEGIN
  DBMS_WORKLOAD_REPOSITORY.DROP_BASELINE (baseline_name => 'peak baseline',
                  cascade => FALSE, dbid => 3310949047);
END;
/
```

   In the example, the name of baseline is peak baseline. The cascade parameter is set to FALSE, which specifies that only the baseline is dropped. Setting this parameter to TRUE specifies that the drop operation will also remove the snapshots associated with the baseline. The optional dbid parameter specifies the database identifier, which in this example is 3310949047. If you do not specify a value for dbid, the local database identifier is used as the default value.

Renaming a Baseline

This section describes how to rename a baseline.

To rename a baseline:

1. Review the existing baselines in the DBA_HIST_BASELINE view to determine the baseline that you want to rename.

2. Use the RENAME_BASELINE procedure to rename the desired baseline:

```
BEGIN
    DBMS_WORKLOAD_REPOSITORY.RENAME_BASELINE (
                    old_baseline_name => 'peak baseline',
                    new_baseline_name => 'peak mondays',
                    dbid => 3310949047);
END;
/
```

   In this example, the name of the baseline is renamed from peak baseline, as specified by the old_baseline_name parameter, to peak mondays, as specified by the new_baseline_name parameter. The optional dbid parameter specifies the database identifier, which in this example is 3310949047. If you do not specify a value for dbid, the local database identifier is used as the default value.

Displaying Baseline Metrics

This section describes how to display metric threshold settings during the time period captured in a baseline. When used with adaptive thresholds, a baseline contains AWR data that can be used to compute metric threshold values. The SELECT_BASELINE_METRICS function enables you to display the summary statistics for metric values in a baseline period.

To display metric information in a baseline:

1. Review the existing baselines in the DBA_HIST_BASELINE view to determine the baseline for which you want to display metric information.
2. Use the SELECT_BASELINE_METRICS function to display the metric information for the desired baseline:

```
BEGIN
    DBMS_WORKLOAD_REPOSITORY.SELECT_BASELINE_METRICS (
                    baseline_name => 'peak baseline',
                    dbid => 3310949047,
                    instance_num => '1');
END;
/
```

   In this example, the name of baseline is peak baseline. The optional dbid parameter specifies the database identifier, which in this example is 3310949047. If you do not specify a value for dbid, the local database identifier is used as the default value. The optional instance_num parameter specifies the instance number, which in this example is 1. If you do not specify a value for instance_num, the local instance is used as the default value.

Modifying the Window Size of the Default Moving Window Baseline

This section describes how to modify the window size of the default moving window baseline. For information about the default moving window baseline, see "Moving Window Baseline" on page 5-10.

To resize the default moving window baseline, use the MODIFY_BASELINE_WINDOW_SIZE procedure:

```
BEGIN
    DBMS_WORKLOAD_REPOSITORY.MODIFY_BASELINE_WINDOW_SIZE (
                    window_size => 30,
                    dbid => 3310949047);
```

```
END;
/
```

The `window_size` parameter is used to specify the new window size, in number of days, for the default moving window size. In this example, the `window_size` parameter is set to 30. The window size must be set to a value that is equal to or less than the value of the AWR retention setting. To set a window size that is greater than the current AWR retention period, you will first need to increase the value of the `retention` parameter, as described in "Modifying Snapshot Settings" on page 5-13.

In this example, the optional dbid parameter specifies the database identifier is `3310949047`. If you do not specify a value for `dbid`, the local database identifier is used as the default value.

Managing Baseline Templates

This section describes how to manage baseline templates. You can automatically create baselines to capture specified time periods in the future using baseline templates. For information about baseline templates, see "Baseline Templates" on page 5-10.

The primary interface for managing baseline templates is Oracle Enterprise Manager. Whenever possible, you should manage baseline templates using Oracle Enterprise Manager, as described in *Oracle Database 2 Day + Performance Tuning Guide*. If Oracle Enterprise Manager is unavailable, you can manage baseline templates using the `DBMS_WORKLOAD_REPOSITORY` package, as described in the following sections:

This section contains the following topics:

- Creating a Single Baseline Template
- Creating a Repeating Baseline Template
- Dropping a Baseline Template

> **See Also:** *Oracle Database PL/SQL Packages and Types Reference* for detailed information on the `DBMS_WORKLOAD_REPOSITORY` package

Creating a Single Baseline Template

This section describes how to create a single baseline template. A single baseline template can be used to create a baseline during a single, fixed time interval in the future. For example, you can create a single baseline template to generate a baseline that is captured on April 2, 2007 from 5:00 p.m. to 8:00 p.m.

To create a single baseline template, use the `CREATE_BASELINE_TEMPLATE` procedure:

```
BEGIN
    DBMS_WORKLOAD_REPOSITORY.CREATE_BASELINE_TEMPLATE (
                    start_time => '2007-04-02 17:00:00 PST',
                    end_time => '2007-04-02 20:00:00 PST',
                    baseline_name => 'baseline_070402',
                    template_name => 'template_070402', expiration => 30,
                    dbid => 3310949047);
END;
/
```

The `start_time` parameter specifies the start time for the baseline to be created. The `end_time` parameter specifies the end time for the baseline to be created. The `baseline_name` parameter specifies the name of the baseline to be created. The

template_name parameter specifies the name of the baseline template. The optional expiration parameter specifies the expiration, in number of days, for the baseline. If unspecified, the baseline will never expire. The optional dbid parameter specifies the database identifier. If unspecified, the local database identifier is used as the default value.

In this example, a baseline template named template_070402 is created that will generate a baseline named baseline_070402 for the time period from 5:00 p.m. to 8:00 p.m. on April 2, 2007 on the database with a database Id of 3310949047. The baseline will expire after 30 days.

Creating a Repeating Baseline Template

This section describes how to create a repeating baseline template. A repeating baseline template can be used to automatically create baselines that repeat during a particular time interval over a specific period in the future. For example, you can create a repeating baseline template to generate a baseline that repeats every Monday from 5:00 p.m. to 8:00 p.m. for the year 2007.

To create a repeating baseline template, use the CREATE_BASELINE_TEMPLATE procedure:

```
BEGIN
    DBMS_WORKLOAD_REPOSITORY.CREATE_BASELINE_TEMPLATE (
                  day_of_week => 'monday', hour_in_day => 17,
                  duration => 3, expiration => 30,
                  start_time => '2007-04-02 17:00:00 PST',
                  end_time => '2007-12-31 20:00:00 PST',
                  baseline_name_prefix => 'baseline_2007_mondays_',
                  template_name => 'template_2007_mondays',
                  dbid => 3310949047);
END;
/
```

The day_of_week parameter specifies the day of the week on which the baseline will repeat. The hour_in_day parameter specifies the hour in the day when the baseline will start. The duration parameter specifies the duration, in number of hours, that the baseline will last. The expiration parameter specifies the number of days to retain each created baseline. If set to NULL, the baselines will never expire. The start_time parameter specifies the start time for the baseline to be created. The end_time parameter specifies the end time for the baseline to be created. The baseline_name_prefix parameter specifies the name of the baseline prefix that will be appended to the data information when the baseline is created. The template_name parameter specifies the name of the baseline template. The optional dbid parameter specifies the database identifier. If unspecified, the local database identifier is used as the default value.

In this example, a baseline template named template_2007_mondays is created that will generate a baseline on every Monday from 5:00 p.m. to 8:00 p.m. beginning on April 2, 2007 at 5:00 p.m. and ending on December 31, 2007 at 8:00 p.m. on the database with a database Id of 3310949047. Each of the baselines will be created with a baseline name with the prefix baseline_2007_mondays_ and will expire after 30 days.

Dropping a Baseline Template

This section describes how to drop an existing baseline template. Periodically, you may want to remove baselines templates that are no longer used to conserve disk space.

To drop a baseline template:

1. Review the existing baselines in the DBA_HIST_BASELINE_TEMPLATE view to determine the baseline template you want to drop.
2. Use the DROP_BASELINE_TEMPLATE procedure to drop the desired baseline template:

```
BEGIN
  DBMS_WORKLOAD_REPOSITORY.DROP_BASELINE_TEMPLATE (
                  template_name => 'template_2007_mondays',
                  dbid => 3310949047);
END;
/
```

   The template_name parameter specifies the name of the baseline template that will be dropped. In the example, the name of baseline template that will be dropped is template_2007_mondays. The optional dbid parameter specifies the database identifier, which in this example is 3310949047. If you do not specify a value for dbid, the local database identifier is used as the default value.

Transporting Automatic Workload Repository Data

Oracle Database enables you to transport AWR data between systems. This is useful in cases where you want to use a separate system to perform analysis of the AWR data. To transport AWR data, you need to first extract the AWR snapshot data from the database on the source system, then load the data into the database on the target system, as described in the following sections:

- Extracting AWR Data
- Loading AWR Data

Extracting AWR Data

The awrextr.sql script extracts the AWR data for a range of snapshots from the database into a Data Pump export file. Once created, this dump file can be transported to another system where the extracted data can be loaded. To run the awrextr.sql script, you need to be connected to the database as the SYS user.

To extract AWR data:

1. At the SQL prompt, enter:

```
@$ORACLE_HOME/rdbms/admin/awrextr.sql
```

   A list of the databases in the AWR schema is displayed.

2. Specify the database from which the AWR data will be extracted:

```
Enter value for db_id: 1377863381
```

   In this example, the database with the database identifier of 1377863381 is selected.

3. Specify the number of days for which you want to list snapshot Ids.

```
Enter value for num_days: 2
```

   A list of existing snapshots for the specified time range is displayed. In this example, snapshots captured in the last 2 days are displayed.

4. Define the range of snapshots for which AWR data will be extracted by specifying a beginning and ending snapshot Id:

```
Enter value for begin_snap: 30
```

```
Enter value for end_snap: 40
```

In this example, the snapshot with a snapshot Id of 30 is selected as the beginning snapshot, and the snapshot with a snapshot Id of 40 is selected as the ending snapshot.

5. A list of directory objects is displayed.

   Specify the directory object pointing to the directory where the export dump file will be stored:

   ```
   Enter value for directory_name: DATA_PUMP_DIR
   ```

   In this example, the directory object `DATA_PUMP_DIR` is selected.

6. Specify the prefix for name of the export dump file (the .dmp suffix will be automatically appended):

   ```
   Enter value for file_name: awrdata_30_40
   ```

   In this example, an export dump file named awrdata_30_40 will be created in the directory corresponding to the directory object you specified:

   ```
   Dump file set for SYS.SYS_EXPORT_TABLE_01 is:
   C:\ORACLE\PRODUCT\11.1.0.5\DB_1\RDBMS\LOG\AWRDATA_30_40.DMP
   Job "SYS"."SYS_EXPORT_TABLE_01" successfully completed at 08:58:20
   ```

   Depending on the amount of AWR data that needs to be extracted, the AWR extract operation may take a while to complete. Once the dump file is created, you can use Data Pump to transport the file to another system.

   > **See Also:** *Oracle Database Utilities* for information about using Data Pump

Loading AWR Data

Once the export dump file is transported to the target system, you can load the extracted AWR data using the `awrload.sql` script. The `awrload.sql` script will first create a staging schema where the snapshot data is transferred from the Data Pump file into the database. The data is then transferred from the staging schema into the appropriate AWR tables. To run the `awrload.sql` script, you need to be connected to the database as the `SYS` user.

To load AWR data:

1. At the SQL prompt, enter:

   ```
   @$ORACLE_HOME/rdbms/admin/awrload.sql
   ```

   A list of directory objects is displayed.

2. Specify the directory object pointing to the directory where the export dump file is located:

   ```
   Enter value for directory_name: DATA_PUMP_DIR
   ```

   In this example, the directory object `DATA_PUMP_DIR` is selected.

3. Specify the prefix for name of the export dump file (the .dmp suffix will be automatically appended):

   ```
   Enter value for file_name: awrdata_30_40
   ```

   In this example, the export dump file named awrdata_30_40 is selected.

4. Specify the name of the staging schema where the AWR data will be loaded:

   ```
   Enter value for schema_name: AWR_STAGE
   ```

   In this example, a staging schema named AWR_STAGE will be created where the AWR data will be loaded.

5. Specify the default tablespace for the staging schema:

   ```
   Enter value for default_tablespace: SYSAUX
   ```

   In this example, the SYSAUX tablespace is selected.

6. Specify the temporary tablespace for the staging schema:

   ```
   Enter value for temporary_tablespace: TEMP
   ```

   In this example, the TEMP tablespace is selected.

7. A staging schema named AWR_STAGE will be created where the AWR data will be loaded. After the AWR data is loaded into the AWR_STAGE schema, the data will be transferred into the AWR tables in the SYS schema:

   ```
   Processing object type TABLE_EXPORT/TABLE/CONSTRAINT/CONSTRAINT
   Completed 113 CONSTRAINT objects in 11 seconds
   Processing object type TABLE_EXPORT/TABLE/CONSTRAINT/REF_CONSTRAINT
   Completed 1 REF_CONSTRAINT objects in 1 seconds
   Job "SYS"."SYS_IMPORT_FULL_03" successfully completed at 09:29:30
   ... Dropping AWR_STAGE user
   End of AWR Load
   ```

   Depending on the amount of AWR data that needs to be loaded, the AWR load operation may take a while to complete. After the AWR data is loaded, the staging schema will be dropped automatically.

Using Automatic Workload Repository Views

Typically, you would view the AWR data through Oracle Enterprise Manager screens or AWR reports. However, you can view the statistics with the following views:

- V$ACTIVE_SESSION_HISTORY

  This view displays active database session activity, sampled once every second. See "Active Session History" on page 5-3.

- V$ metric views provide metric data to track the performance of the system

  The metric views are organized into various groups, such as event, event class, system, session, service, file, and tablespace metrics. These groups are identified in the V$METRICGROUP view.

- DBA_HIST views

  The DBA_HIST views displays historical data stored in the database. This group of views includes:

  - DBA_HIST_ACTIVE_SESS_HISTORY displays the history of the contents of the in-memory active session history for recent system activity
  - DBA_HIST_BASELINE displays information about the baselines captured on the system, such as the time range of each baseline and the baseline type
  - DBA_HIST_BASELINE_DETAILS displays details about a specific baseline

- DBA_HIST_BASELINE_TEMPLATE displays information about the baseline templates used by the system to generate baselines
- DBA_HIST_DATABASE_INSTANCE displays information about the database environment
- DBA_HIST_SNAPSHOT displays information on snapshots in the system
- DBA_HIST_SQL_PLAN displays the SQL execution plans
- DBA_HIST_WR_CONTROL displays the settings for controlling AWR

See Also: *Oracle Database Reference* for information on dynamic and static data dictionary views

Generating Automatic Workload Repository Reports

An AWR report shows data captured between two snapshots (or two points in time). The AWR reports are divided into multiple sections. The HTML report includes links that can be used to navigate quickly between sections. The content of the report contains the workload profile of the system for the selected range of snapshots.

The primary interface for generating AWR reports is Oracle Enterprise Manager. Whenever possible, you should generate AWR reports using Oracle Enterprise Manager, as described in *Oracle Database 2 Day + Performance Tuning Guide*. If Oracle Enterprise Manager is unavailable, you can generate AWR reports by running SQL scripts, as described in the following sections:

- Generating an AWR Report for a Snapshot Range
- Generating an AWR Report for a Snapshot Range on a Specified Database Instance
- Generating an AWR Report for a SQL Statement
- Generating an AWR Report for a SQL Statement on a Specified Database Instance

To run these scripts, you must be granted the DBA role.

Note: If you run a report on a database that does not have any workload activity during the specified range of snapshots, calculated percentages for some report statistics can be less than 0 or greater than 100. This result simply means that there is no meaningful value for the statistic.

Generating an AWR Report for a Snapshot Range

The `awrrpt.sql` SQL script generates an HTML or text report that displays statistics for a range of snapshot Ids.

To generate an AWR report for a range of snapshots:

1. At the SQL prompt, enter:

```
@$ORACLE_HOME/rdbms/admin/awrrpt.sql
```

2. Specify whether you want an HTML or a text report:

```
Enter value for report_type: text
```

   In this example, a text report is chosen.

3. Specify the number of days for which you want to list snapshot Ids.

```
Enter value for num_days: 2
```

A list of existing snapshots for the specified time range is displayed. In this example, snapshots captured in the last 2 days are displayed.

4. Specify a beginning and ending snapshot Id for the workload repository report:

```
Enter value for begin_snap: 150
Enter value for end_snap: 160
```

In this example, the snapshot with a snapshot Id of 150 is selected as the beginning snapshot, and the snapshot with a snapshot Id of 160 is selected as the ending snapshot.

5. Enter a report name, or accept the default report name:

```
Enter value for report_name:
Using the report name awrrpt_1_150_160
```

In this example, the default name is accepted and an AWR report named awrrpt_1_150_160 is generated.

Generating an AWR Report for a Snapshot Range on a Specified Database Instance

The `awrrpti.sql` SQL script generates an HTML or text report that displays statistics for a range of snapshot Ids on a specified database and instance. This script enables you to specify a database and instance before entering a range of snapshot Ids.

To generate an AWR report for a range of snapshots on a specific database instance:

1. At the SQL prompt, enter:

```
@$ORACLE_HOME/rdbms/admin/awrrpti.sql
```

2. Specify whether you want an HTML or a text report:

```
Enter value for report_type: text
```

In this example, a text report is chosen.

A list of available database identifiers and instance numbers are displayed:

```
Instances in this Workload Repository schema
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
   DB Id    Inst Num DB Name      Instance     Host
------------ -------- ------------ ------------ ------------
 3309173529        1 MAIN         main         dlsun1690
 3309173529        1 TINT251      tint251      stint251
```

3. Enter the values for the database identifier (`dbid`) and instance number (`inst_num`):

```
Enter value for dbid: 3309173529
Using 3309173529 for database Id
Enter value for inst_num: 1
```

4. Specify the number of days for which you want to list snapshot Ids.

```
Enter value for num_days: 2
```

A list of existing snapshots for the specified time range is displayed. In this example, snapshots captured in the last 2 days are displayed.

5. Specify a beginning and ending snapshot Id for the workload repository report:

```
Enter value for begin_snap: 150
Enter value for end_snap: 160
```

In this example, the snapshot with a snapshot Id of 150 is selected as the beginning snapshot, and the snapshot with a snapshot Id of 160 is selected as the ending snapshot.

6. Enter a report name, or accept the default report name:

```
Enter value for report_name:
Using the report name awrrpt_1_150_160
```

In this example, the default name is accepted and an AWR report named awrrpt_1_150_160 is generated on the database instance with a database Id value of 3309173529.

Generating an AWR Report for a SQL Statement

The `awrsqrpt.sql` SQL script generates an HTML or text report that displays statistics of a particular SQL statement for a range of snapshot Ids. Run this report to inspect or debug the performance of a SQL statement.

To generate an AWR report for a particular SQL statement:

1. At the SQL prompt, enter:

```
@$ORACLE_HOME/rdbms/admin/awrsqrpt.sql
```

2. Specify whether you want an HTML or a text report:

```
Enter value for report_type: html
```

In this example, an HTML report is chosen.

3. Specify the number of days for which you want to list snapshot Ids.

```
Enter value for num_days: 1
```

A list of existing snapshots for the specified time range is displayed. In this example, snapshots captured in the previous day are displayed.

4. Specify a beginning and ending snapshot Id for the workload repository report:

```
Enter value for begin_snap: 146
Enter value for end_snap: 147
```

In this example, the snapshot with a snapshot Id of 146 is selected as the beginning snapshot, and the snapshot with a snapshot Id of 147 is selected as the ending snapshot.

5. Specify the SQL Id of a particular SQL statement to display statistics:

```
Enter value for sql_id: 2b064ybzkwf1y
```

In this example, the SQL statement with a SQL Id of 2b064ybzkwf1y is selected.

6. Enter a report name, or accept the default report name:

```
Enter value for report_name:
Using the report name awrrpt_1_146_147.html
```

In this example, the default name is accepted and an AWR report named awrrpt_1_146_147 is generated.

Generating an AWR Report for a SQL Statement on a Specified Database Instance

The `awrsqrpi.sql` SQL script generates an HTML or text report that displays statistics of a particular SQL statement for a range of snapshot Ids on a specified database and instance. This script enables you to specify a database and instance before selecting a SQL statement. Run this report to inspect or debug the performance of a SQL statement on a specific database and instance.

To generate an AWR report for a particular SQL statement on a specified database instance:

1. At the SQL prompt, enter:

   ```
   @$ORACLE_HOME/rdbms/admin/awrsqrpi.sql
   ```

2. Specify whether you want an HTML or a text report:

   ```
   Enter value for report_type: html
   ```

   In this example, an HTML report is chosen.

   A list of available database identifiers and instance numbers are displayed:

   ```
   Instances in this Workload Repository schema
   ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
      DB Id    Inst Num DB Name      Instance     Host
   ----------- -------- ------------ ------------ ------------
    3309173529        1 MAIN         main         dlsun1690
    3309173529        1 TINT251      tint251      stint251
   ```

3. Enter the values for the database identifier (`dbid`) and instance number (`inst_num`):

   ```
   Enter value for dbid: 3309173529
   Using 3309173529 for database Id
   Enter value for inst_num: 1
   Using 1 for instance number
   ```

4. Specify the number of days for which you want to list snapshot Ids.

   ```
   Enter value for num_days: 1
   ```

   A list of existing snapshots for the specified time range is displayed. In this example, snapshots captured in the previous day are displayed.

5. Specify a beginning and ending snapshot Id for the workload repository report:

   ```
   Enter value for begin_snap: 146
   Enter value for end_snap: 147
   ```

   In this example, the snapshot with a snapshot Id of 146 is selected as the beginning snapshot, and the snapshot with a snapshot Id of 147 is selected as the ending snapshot.

6. Specify the SQL Id of a particular SQL statement to display statistics:

   ```
   Enter value for sql_id: 2b064ybzkwf1y
   ```

   In this example, the SQL statement with a SQL Id of 2b064ybzkwf1y is selected.

7. Enter a report name, or accept the default report name:

   ```
   Enter value for report_name:
   Using the report name awrrpt_1_146_147.html
   ```

In this example, the default name is accepted and an AWR report named awrrpt_1_146_147 is generated on the database instance with a database Id value of 3309173529.

Generating Automatic Workload Repository Compare Periods Reports

While an AWR report shows AWR data between two snapshots (or two points in time), the AWR Compare Periods report shows the difference between two periods (or two AWR reports, which equates to four snapshots). Using the AWR Compare Periods report helps you to identify detailed performance attributes and configuration settings that differ between two time periods. For example, if the application workload is known to be stable between 10:00 p.m. and midnight every night, but the performance on a particular Thursday was poor between 10:00 p.m. and 11:00 p.m., generating an AWR Compare Periods report for Thursday from 10:00 p.m. to 11:00 p.m. and Wednesday from 10:00 p.m. to 11:00 p.m. should identify configuration settings, workload profile, and statistics that were different in these two time periods. Based on the differences identified, the cause of the performance degradation can be more easily diagnosed. The two time periods selected for the AWR Compare Periods Report can be of different durations, because the report normalizes the statistics by the amount of time spent on the database for each time period, and presents statistical data ordered by the largest difference between the periods.

The AWR Compare Periods reports are divided into multiple sections. The HTML report includes links that can be used to navigate quickly between sections. The content of the report contains the workload profile of the system for the selected range of snapshots.

The primary interface for generating AWR Compare Periods reports is Oracle Enterprise Manager. Whenever possible, you should generate AWR Compare Periods reports using Oracle Enterprise Manager, as described in *Oracle Database 2 Day + Performance Tuning Guide*. If Oracle Enterprise Manager is unavailable, you can generate AWR Compare Periods reports by running SQL scripts, as described in the following sections:

- Generating an AWR Compare Periods Report
- Generating an AWR Compare Periods Report on a Specified Database Instance

To run these scripts, you must be granted the DBA role.

Generating an AWR Compare Periods Report

The `awrddrpt.sql` SQL script generates an HTML or text report that compares detailed performance attributes and configuration settings between two selected time periods.

To generate an AWR Compare Periods report:

1. At the SQL prompt, enter:

   ```
   @$ORACLE_HOME/rdbms/admin/awrddrpt.sql
   ```

2. Specify whether you want an HTML or a text report:

   ```
   Enter value for report_type: html
   ```

   In this example, an HTML report is chosen.

3. Specify the number of days for which you want to list snapshot Ids in the first time period.

   ```
   Enter value for num_days: 2
   ```

A list of existing snapshots for the specified time range is displayed. In this example, snapshots captured in the last 2 days are displayed.

4. Specify a beginning and ending snapshot Id for the first time period:

```
Enter value for begin_snap: 102
Enter value for end_snap: 103
```

In this example, the snapshot with a snapshot Id of 102 is selected as the beginning snapshot, and the snapshot with a snapshot Id of 103 is selected as the ending snapshot for the first time period.

5. Specify the number of days for which you want to list snapshot Ids in the second time period.

```
Enter value for num_days2: 1
```

A list of existing snapshots for the specified time range is displayed. In this example, snapshots captured in the previous day are displayed.

6. Specify a beginning and ending snapshot Id for the second time period:

```
Enter value for begin_snap2: 126
Enter value for end_snap2: 127
```

In this example, the snapshot with a snapshot Id of 126 is selected as the beginning snapshot, and the snapshot with a snapshot Id of 127 is selected as the ending snapshot for the second time period.

7. Enter a report name, or accept the default report name:

```
Enter value for report_name:
Using the report name awrdiff_1_102_1_126.txt
```

In this example, the default name is accepted and an AWR report named awrdiff_1_102_126 is generated.

Generating an AWR Compare Periods Report on a Specified Database Instance

The `awrddrpi.sql` SQL script generates an HTML or text report that compares detailed performance attributes and configuration settings between two selected time periods on a specific database and instance. This script enables you to specify a database and instance before selecting time periods to compare.

To generate an AWR Compare Periods report on a specified database instance:

1. At the SQL prompt, enter:

```
@$ORACLE_HOME/rdbms/admin/awrddrpi.sql
```

2. Specify whether you want an HTML or a text report:

```
Enter value for report_type: text
```

In this example, a text report is chosen.

3. A list of available database identifiers and instance numbers are displayed:

```
Instances in this Workload Repository schema
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
   DB Id    Inst Num DB Name      Instance     Host
------------ -------- ------------ ------------ ------------
 3309173529        1 MAIN         main         dlsun1690
```

```
 3309173529        1 TINT251      tint251      stint251
```

Enter the values for the database identifier (`dbid`) and instance number (`inst_num`) for the first time period:

```
Enter value for dbid: 3309173529
Using 3309173529 for Database Id for the first pair of snapshots
Enter value for inst_num: 1
Using 1 for Instance Number for the first pair of snapshots
```

4. Specify the number of days for which you want to list snapshot Ids in the first time period.

```
Enter value for num_days: 2
```

A list of existing snapshots for the specified time range is displayed. In this example, snapshots captured in the last 2 days are displayed.

5. Specify a beginning and ending snapshot Id for the first time period:

```
Enter value for begin_snap: 102
Enter value for end_snap: 103
```

In this example, the snapshot with a snapshot Id of 102 is selected as the beginning snapshot, and the snapshot with a snapshot Id of 103 is selected as the ending snapshot for the first time period.

6. Enter the values for the database identifier (`dbid`) and instance number (`inst_num`) for the second time period:

```
Enter value for dbid2: 3309173529
Using 3309173529 for Database Id for the second pair of snapshots
Enter value for inst_num2: 1
Using 1 for Instance Number for the second pair of snapshots
```

7. Specify the number of days for which you want to list snapshot Ids in the second time period.

```
Enter value for num_days2: 1
```

A list of existing snapshots for the specified time range is displayed. In this example, snapshots captured in the previous day are displayed.

8. Specify a beginning and ending snapshot Id for the second time period:

```
Enter value for begin_snap2: 126
Enter value for end_snap2: 127
```

In this example, the snapshot with a snapshot Id of 126 is selected as the beginning snapshot, and the snapshot with a snapshot Id of 127 is selected as the ending snapshot for the second time period.

9. Enter a report name, or accept the default report name:

```
Enter value for report_name:
Using the report name awrdiff_1_102_1_126.txt
```

In this example, the default name is accepted and an AWR report named awrdiff_1_102_126 is generated on the database instance with a database Id value of 3309173529.

Generating Active Session History Reports

Use Active Session History (ASH) reports to perform analysis of:

- Transient performance problems that typically last for a few minutes
- Scoped or targeted performance analysis by various dimensions or their combinations, such as time, session, module, action, or `SQL_ID`

Transient performance problems are short-lived and do not appear in the Automatic Database Diagnostics Monitor (ADDM) analysis. ADDM tries to report the most significant performance problems during an analysis period in terms of their impact on DB time. If a particular problem lasts for a very short duration, its severity might be averaged out or minimized by other performance problems in the entire analysis period; therefore, the problem may not appear in the ADDM findings. Whether or not a performance problem is captured by ADDM depends on its duration compared to the interval between the Automatic Workload Repository (AWR) snapshots.

If a performance problem lasts for a significant portion of the time between snapshots, it will be captured by ADDM. For example, if the snapshot interval is set to one hour, a performance problem that lasts for 30 minutes should not be considered as a transient performance problem because its duration represents a significant portion of the snapshot interval and will likely be captured by ADDM.

On the other hand, a performance problem that lasts for only 2 minutes could be a transient performance problem because its duration represents a small portion of the snapshot interval and will likely not show up in the ADDM findings. For example, if the user notifies you that the system was slow between 10:00 p.m. and 10:10 p.m., but the ADDM analysis for the time period between 10:00 p.m. and 11:00 p.m. does not show a performance problem, it is likely that a transient performance problem occurred that lasted for only a few minutes of the 10-minute interval reported by the user.

The ASH reports are divided into multiple sections. The HTML report includes links that can be used to navigate quickly between sections. The content of the report contains ASH information used to identify blocker and waiter identities and their associated transaction identifiers and SQL for a specified duration. For more information on ASH, see "Active Session History" on page 5-3.

The primary interface for generating ASH reports is Oracle Enterprise Manager. Whenever possible, you should generate ASH reports using Oracle Enterprise Manager, as described in *Oracle Database 2 Day + Performance Tuning Guide*. If Oracle Enterprise Manager is unavailable, you can generate ASH reports by running SQL scripts, as described in the following sections:

- Generating an ASH Report
- Generating an ASH Report on a Specified Database Instance

Generating an ASH Report

The `ashrpt.sql` SQL script generates an HTML or text report that displays ASH information for a specified duration.

To generate an ASH report:

1. At the SQL prompt, enter:

```
@$ORACLE_HOME/rdbms/admin/ashrpt.sql
```

2. Specify whether you want an HTML or a text report:

```
Enter value for report_type: text
```

In this example, a text report is chosen.

3. Specify the begin time in minutes prior to the system date:

```
Enter value for begin_time: -10
```

In this example, 10 minutes before the current time is selected.

4. Enter the duration in minutes that the report for which you want to capture ASH information from the begin time.

```
Enter value for duration:
```

In this example, the default duration of system date minus begin time is accepted.

5. Enter a report name, or accept the default report name:

```
Enter value for report_name:
Using the report name ashrpt_1_0310_0131.txt
```

In this example, the default name is accepted and an ASH report named ashrpt_1_0310_0131 is generated. The report will gather ASH information beginning from 10 minutes before the current time and ending at the current time.

Generating an ASH Report on a Specified Database Instance

The `ashrpti.sql` SQL script generates an HTML or text report that displays ASH information for a specified duration for a specified database and instance. This script enables you to specify a database and instance before setting the time frame to collect ASH information.

To generate an ASH report on a specified database instance:

1. At the SQL prompt, enter:

```
@$ORACLE_HOME/rdbms/admin/ashrpti.sql
```

2. Specify whether you want an HTML or a text report:

```
Enter value for report_type: html
```

In this example, an HTML report is chosen.

3. A list of available database Ids and instance numbers are displayed:

```
Instances in this Workload Repository schema
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
   DB Id    Inst Num DB Name      Instance     Host
------------ -------- ------------ ------------ ------------
 3309173529        1 MAIN         main         dlsun1690
 3309173529        1 TINT251      tint251      stint251
```

Enter the values for the database identifier (`dbid`) and instance number (`inst_num`):

```
Enter value for dbid: 3309173529
Using 3309173529 for database id
Enter value for inst_num: 1
```

4. Specify the begin time in minutes prior to the system date:

```
Enter value for begin_time: -10
```

In this example, 10 minutes before the current time is selected.

5. Enter the duration in minutes that the report for which you want to capture ASH information from the begin time.

```
Enter value for duration:
```

   In this example, the default duration of system date minus begin time is accepted.

6. Enter a report name, or accept the default report name:

```
Enter value for report_name:
Using the report name ashrpt_1_0310_0131.txt
```

   In this example, the default name is accepted and an ASH report named ashrpt_1_0310_0131 is generated. The report will gather ASH information on the database instance with a database Id value of 3309173529 beginning from 10 minutes before the current time and ending at the current time.

Using Active Session History Reports

After generating an ASH report, you can review the contents to identify transient performance problems.

The contents of the ASH report are divided into the following sections:

- Top Events
- Load Profile
- Top SQL
- Top PL/SQL
- Top Java
- Top Sessions
- Top Objects/Files/Latches
- Activity Over Time

> **See Also:** *Oracle Database Oracle Clusterware and Oracle Real Application Clusters Administration and Deployment Guide* for information about sections in the ASH report that are specific to Oracle Real Application Clusters (RAC)

Top Events

The Top Events section describes the top wait events of the sampled session activity categorized by user, background, and priority. Use the information in this section to identify the wait events that may be the cause of the transient performance problem.

The Top Events section contains the following subsections:

- Top User Events
- Top Background Events
- Top Event P1/P2/P3 Values

Top User Events The Top User Events subsection lists the top wait events from user processes that accounted for the highest percentages of sampled session activity.

Top Background Events The Top Background Events subsection lists the top wait events from backgrounds that accounted for the highest percentages of sampled session activity.

Top Event P1/P2/P3 Values The Top Event P1/P2/P3 subsection lists the wait event parameter values of the top wait events that accounted for the highest percentages of sampled session activity, ordered by the percentage of total wait time (% Event). For each wait event, values in the P1 Value, P2 Value, P3 Value column correspond to wait event parameters displayed in the Parameter 1, Parameter 2, and Parameter 3 columns.

Load Profile

The Load Profile section describes the load analyzed in the sampled session activity. Use the information in this section to identify the service, client, or SQL command type that may be the cause of the transient performance problem.

The Load Profile section contains the following subsections:

- Top Service/Module
- Top Client IDs
- Top SQL Command Types
- Top Phases of Execution

Top Service/Module The Top Service/Module subsection lists the services and modules that accounted for the highest percentages of sampled session activity.

Top Client IDs The Top Client IDs subsection lists the clients that accounted for the highest percentages of sampled session activity based on their client ID, which is the application-specific identifier of the database session.

Top SQL Command Types The Top SQL Command Types subsection lists the SQL command types, such as `SELECT`, `UPDATE`, `INSERT`, and `DELETE`, that accounted for the highest percentages of sampled session activity.

Top Phases of Execution The Top Phases of Execution subsection lists the phases of execution, such as SQL, PL/SQL, and Java compilation and execution, that accounted for the highest percentages of sampled session activity.

Top SQL

The Top SQL section describes the top SQL statements of the sampled session activity. Use this information to identify high-load SQL statements that may be the cause of the transient performance problem.

The Top SQL section contains the following subsections:

- Top SQL with Top Events
- Top SQL with Top Row Sources
- Top SQL Using Literals
- Top Parsing Module/Action
- Complete List of SQL Text

Top SQL with Top Events The Top SQL with Top Events subsection lists the SQL statements that accounted for the highest percentages of sampled session activity and

the top wait events that were encountered by these SQL statements. The Sampled # of Executions column shows how many distinct executions of a particular SQL statement were sampled.

Top SQL with Top Row Sources The Top SQL with Top Row Sources subsection lists the SQL statements that accounted for the highest percentages of sampled session activity and their detailed execution plan information. You can use this information to identify which part of the SQL execution contributed significantly to the SQL elapsed time.

Top SQL Using Literals The Top SQL Using Literals subsection lists the SQL statements using literals that accounted for the highest percentages of sampled session activity. You should review the statements listed in this report to determine whether the literals can be replaced with bind variables.

Top Parsing Module/Action The Top Parsing Module/Action subsection lists the module and action that accounted for the highest percentages of sampled session activity while parsing the SQL statement.

Complete List of SQL Text The Complete List of SQL Text subsection displays the entire text of the Top SQL statements shown in this section.

Top PL/SQL

The Top PL/SQL section lists the PL/SQL procedures that accounted for the highest percentages of sampled session activity. The PL/SQL Entry Subprogram column lists the application's top-level entry point into PL/SQL. The PL/SQL Current Subprogram column lists the PL/SQL subprogram being executed at the point of sampling. If the value of this column is `SQL`, then the % Current column shows the percentage of time spent executing SQL for this subprogram.

Top Java

The Top Java section describes the top Java programs in the sampled session activity.

Top Sessions

The Top Sessions section describes the sessions that were waiting for a particular wait event. Use this information to identify the sessions that accounted for the highest percentages of sampled session activity, which may be the cause of the transient performance problem.

The Top Sessions section contains the following subsections:

- Top Sessions
- Top Blocking Sessions
- Top Sessions Running PQs

Top Sessions The Top Session subsection lists the sessions that were waiting for a particular wait event that accounted for the highest percentages of sampled session activity.

Top Blocking Sessions The Top Blocking Sessions subsection lists the blocking sessions that accounted for the highest percentages of sampled session activity.

Top Sessions Running PQs The Top Sessions Running PQs subsection lists the sessions running parallel queries (PQs) that were waiting for a particular wait event, which accounted for the highest percentages of sampled session activity.

Top Objects/Files/Latches

The Top Objects/Files/Latches section provides additional information about the most commonly-used database resources and contains the following subsections:

- Top DB Objects
- Top DB Files
- Top Latches

Top DB Objects The Top DB Objects subsection lists the database objects (such as tables and indexes) that accounted for the highest percentages of sampled session activity.

Top DB Files The Top DB Files subsection lists the database files that accounted for the highest percentages of sampled session activity.

Top Latches The Top Latches subsection lists the latches that accounted for the highest percentages of sampled session activity.

Latches are simple, low-level serialization mechanisms to protect shared data structures in the System Global Area (SGA). For example, latches protect the list of users currently accessing the database and the data structures describing the blocks in the buffer cache. A server or background process acquires a latch for a very short time while manipulating or looking at one of these structures. The implementation of latches is operating system-dependent, particularly in regard to whether and how long a process will wait for a latch.

Activity Over Time

The Activity Over Time section is one of the most informative sections of the ASH report. This section is particularly useful for longer time periods because it provides in-depth details about activities and workload profiles during the analysis period. The Activity Over Time section is divided into 10 time slots. The size of each time slot varies based on the duration of the analysis period. The first and last slots are usually odd-sized. All inner slots are equally sized and can be compared to each other. For example, if the analysis period lasts for 10 minutes, then all time slots will 1 minute each. On the other hand, if the analysis period lasts for 9 minutes and 30 seconds, then the outer slots may be 15 seconds each and the inner slots will be 1 minute each.

Each of the time slots contains information regarding that particular time slot, as described in Table 5–2.

Table 5–2 Activity Over Time

Column	Description
Slot Time (Duration)	Duration of the slot
Slot Count	Number of sampled sessions in the slot
Event	Top three wait events in the slot
Event Count	Number of ASH samples waiting for the wait event
% Event	Percentage of ASH samples waiting for wait events in the entire analysis period

When comparing the inner slots, perform a skew analysis by identifying spikes in the Event Count and Slot Count columns. A spike in the Event Count column indicates an increase in the number of sampled sessions waiting for a particular event. A spike in the Slot Count column indicates an increase in active sessions, because ASH data is

sampled from active sessions only and a relative increase in database workload. Typically, when the number of active session samples and the number of sessions associated with a wait event increases, the slot may be the cause of the transient performance problem.

To generate the ASH report with a user-defined slot size, run the `ashrpti.sql` script, as described in "Generating an ASH Report on a Specified Database Instance" on page 5-29.

6

Automatic Performance Diagnostics

This chapter describes Oracle automatic features for performance diagnosing and tuning.

This chapter contains the following topics:

- Overview of the Automatic Database Diagnostic Monitor
- Setting Up ADDM
- Diagnosing Database Performance Problems with ADDM
- Views with ADDM Information

> **See Also:** *Oracle Database 2 Day + Performance Tuning Guide* for information about using Oracle Enterprise Manager to diagnose and tune the database with the Automatic Database Diagnostic Monitor

Overview of the Automatic Database Diagnostic Monitor

When problems occur with a system, it is important to perform accurate and timely diagnosis of the problem before making any changes to a system. Oftentimes, a database administrator (DBA) simply looks at the symptoms and immediately starts changing the system to fix those symptoms. However, an accurate diagnosis of the actual problem in the initial stage significantly increases the probability of success in resolving the problem.

With Oracle Database, the statistical data needed for accurate diagnosis of a problem is stored in the Automatic Workload Repository (AWR). The Automatic Database Diagnostic Monitor (ADDM):

- Analyzes the AWR data on a regular basis
- Diagnoses the root causes of performance problems
- Provides recommendations for correcting any problems
- Identifies non-problem areas of the system

Because AWR is a repository of historical performance data, ADDM can be used to analyze performance issues after the event, often saving time and resources in reproducing a problem. For information about the AWR, see "Overview of the Automatic Workload Repository" on page 5-8.

In most cases, ADDM output should be the first place that a DBA looks when notified of a performance problem. ADDM provides the following benefits:

- Automatic performance diagnostic report every hour by default
- Problem diagnosis based on decades of tuning expertise

- Time-based quantification of problem impacts and recommendation benefits
- Identification of root cause, not symptoms
- Recommendations for treating the root causes of problems
- Identification of non-problem areas of the system
- Minimal overhead to the system during the diagnostic process

It is important to realize that tuning is an iterative process, and fixing one problem can cause the bottleneck to shift to another part of the system. Even with the benefit of ADDM analysis, it can take multiple tuning cycles to reach acceptable system performance. ADDM benefits apply beyond production systems; on development and test systems, ADDM can provide an early warning of performance issues.

This section contains the following topics:

ADDM Analysis

An ADDM analysis can be performed on a pair of AWR snapshots and a set of instances from the same database. The pair of AWR snapshots define the time period for analysis, and the set of instances define the target for analysis.

If you are using Oracle Real Application Clusters (RAC), ADDM has three analysis modes:

- Database

  In Database mode, ADDM analyzes all instances of the database.

- Instance

  In Instance mode, ADDM analyzes a particular instance of the database.

- Partial

  In Partial mode, ADDM analyzes a subset of all database instances.

If you not using Oracle RAC, ADDM can only function in Instance mode because there is only one instance of the database.

An ADDM analysis is performed each time an AWR snapshot is taken and the results are saved in the database. The time period analyzed by ADDM is defined by the last two snapshots (the last hour by default). ADDM will always analyze the specified instance in Instance mode. For non-Oracle RAC or single instance environments, the analysis performed in the Instance mode is the same as a database-wide analysis. If you are using Oracle RAC, ADDM will also analyze the entire database in Database mode, as described in "Using ADDM with Oracle Real Application Clusters" on page 6-3. After an ADDM completes its analysis, you can view the results using Oracle Enterprise Manager, or by viewing a report in a SQL*Plus session.

ADDM analysis is performed top down, first identifying symptoms, and then refining them to reach the root causes of performance problems. The goal of the analysis is to reduce a single throughput metric called `DB time`. `DB time` is the cumulative time spent by the database in processing user requests. It includes wait time and CPU time

of all non-idle user sessions. DB time is displayed in the V$SESS_TIME_MODEL and V$SYS_TIME_MODEL views.

See Also:

- *Oracle Database Reference* for information about the V$SESS_TIME_MODEL and V$SYS_TIME_MODEL views
- "Time Model Statistics" on page 5-3 for a discussion of time model statistics and DB time
- *Oracle Database Concepts* for information on server processes

By reducing DB time, the database is able to support more user requests using the same resources, which increases throughput. The problems reported by ADDM are sorted by the amount of DB time they are responsible for. System areas that are not responsible for a significant portion of DB time are reported as non-problem areas.

The types of problems that ADDM considers include the following:

- CPU bottlenecks - Is the system CPU bound by Oracle or some other application?
- Undersized Memory Structures - Are the Oracle memory structures, such as the SGA, PGA, and buffer cache, adequately sized?
- I/O capacity issues - Is the I/O subsystem performing as expected?
- High load SQL statements - Are there any SQL statements which are consuming excessive system resources?
- High load PL/SQL execution and compilation, as well as high load Java usage
- RAC specific issues - What are the global cache hot blocks and objects; are there any interconnect latency issues?
- Sub-optimal use of Oracle by the application - Are there problems with poor connection management, excessive parsing, or application level lock contention?
- Database configuration issues - Is there evidence of incorrect sizing of log files, archiving issues, excessive checkpoints, or sub-optimal parameter settings?
- Concurrency issues - Are there buffer busy problems?
- Hot objects and top SQL for various problem areas

Note: This is not a comprehensive list of all problem types that ADDM considers in its analysis.

ADDM also documents the non-problem areas of the system. For example, wait event classes that are not significantly impacting the performance of the system are identified and removed from the tuning consideration at an early stage, saving time and effort that would be spent on items that do not impact overall system performance.

Using ADDM with Oracle Real Application Clusters

If you are using Oracle Real Application Clusters (RAC), you can run ADDM in Database analysis mode to analyze the throughput performance of all instances of the database. In Database mode, ADDM considers DB time as the sum of the database time for all database instances. Using the Database analysis mode enables you to view

all findings that are significant to the entire database in a single report, instead of reviewing a separate report for each instance.

The Database mode report includes findings about database resources (such as I/O and interconnect). The report also aggregates findings from the various instances if they are significant to the entire database. For example, if the CPU load on a single instance is high enough to affect the entire database, the finding will appear in the Database mode analysis, which will point to the particular instance responsible for the problem.

> **See Also:** *Oracle Database 2 Day + Real Application Clusters Guide* for information about using ADDM with Oracle RAC

ADDM Analysis Results

In addition to problem diagnostics, ADDM recommends possible solutions. ADDM analysis results are represented as a set of findings. See Example 6–1 on page 6-5 for an example of ADDM analysis result. Each ADDM finding can belong to one of three types:

- Problem findings describe the root cause of a database performance problem.
- Symptom findings contain information that often lead to one or more problem findings.
- Information findings are used for reporting information that are relevant to understanding the performance of the database, but do not constitute a performance problem (such as non-problem areas of the database and the activity of automatic database maintenance).
- Warning findings contain information about problems that may affect the completeness or accuracy of the ADDM analysis (such as missing data in the AWR)

Each problem finding is quantified by an impact that is an estimate of the portion of `DB time` caused by the finding's performance issue. A problem finding can be associated with a list of recommendations for reducing the impact of the performance problem. The types of recommendations include:

- Hardware changes - Adding CPUs or changing the I/O subsystem configuration
- Database configuration - Changing initialization parameter settings
- Schema changes - Hash partitioning a table or index, or using automatic segment-space management (ASSM)
- Application changes - Using the cache option for sequences or using bind variables
- Using other advisors - Running the SQL Tuning Advisor on high load SQL or running the Segment Advisor on hot objects

A list of recommendations can contain various alternatives for solving the same problem; you do not have to apply all the recommendations to solve a specific problem. Each recommendation has a benefit which is an estimate of the portion of `DB time` that can be saved if the recommendation is implemented. Recommendations are composed of actions and rationales. You need to apply all the actions of a recommendation in order to gain the estimated benefit. The rationales are used for explaining why the set of actions were recommended and to provide additional information to implement the suggested recommendation.

Reviewing ADDM Analysis Results: Example

Consider the following section of an ADDM report in Example 6–1.

Example 6–1 Example ADDM Report

```
FINDING 1: 31% impact (7798 seconds)
-----------------------------------
SQL statements were not shared due to the usage of literals. This resulted in
additional hard parses which were consuming significant database time.

RECOMMENDATION 1: Application Analysis, 31% benefit (7798 seconds)
  ACTION: Investigate application logic for possible use of bind variables
    instead of literals. Alternatively, you may set the parameter
    "cursor_sharing" to "force".
  RATIONALE: SQL statements with PLAN_HASH_VALUE 3106087033 were found to be
    using literals. Look in V$SQL for examples of such SQL statements.
```

In this example, the finding points to a particular root cause, the usage of literals in SQL statements, which is estimated to have an impact of about 31% of total DB time in the analysis period.

The finding has a recommendation associated with it, composed of one action and one rationale. The action specifies a solution to the problem found and is estimated to have a maximum benefit of up to 31% DB time in the analysis period. Note that the benefit is given as a portion of the total DB time and not as a portion of the finding's impact. The rationale provides additional information on tracking potential SQL statements that were using literals and causing this performance issue. Using the specified plan hash value of SQL statements that could be a problem, a DBA could quickly examine a few sample statements.

When a specific problem has multiple causes, the ADDM may report multiple problem and symptom findings. In this case, the impacts of these multiple findings can contain the same portion of DB time. Because the performance issues of findings can overlap, summing all the impacts of the reported findings can yield a number higher than 100% of DB time. For example, if a system performs many read I/Os the ADDM might report a SQL statement responsible for 50% of DB time due to I/O activity as one finding, and an undersized buffer cache responsible for 75% of DB time as another finding.

When multiple recommendations are associated with a problem finding, the recommendations may contain alternatives for solving the problem. In this case, the sum of the recommendations' benefits may be higher than the finding's impact.

When appropriate, an ADDM action many haves multiple solutions for you to choose from. In the example, the most effective solution is to use bind variables. However, it is often difficult to modify the application. Changing the value of the CURSOR_SHARING initialization parameter is much easier to implement and can provide significant improvement.

Setting Up ADDM

Automatic database diagnostic monitoring is enabled by default and is controlled by the CONTROL_MANAGEMENT_PACK_ACCESS and the STATISTICS_LEVEL initialization parameters.

The CONTROL_MANAGEMENT_PACK_ACCESS parameter should be set to DIAGNOSTIC or DIAGNOSTIC+TUNING to enable automatic database diagnostic monitoring. The

default setting is DIAGNOSTIC+TUNING. Setting CONTROL_MANAGEMENT_PACK_ACCESS to NONE disables ADDM.

The STATISTICS_LEVEL parameter should be set to the TYPICAL or ALL to enable automatic database diagnostic monitoring. The default setting is TYPICAL. Setting STATISTICS_LEVEL to BASIC disables many Oracle Database features, including ADDM, and is strongly discouraged.

> **See Also:** *Oracle Database Reference* for information about the CONTROL_MANAGEMENT_PACK_ACCESS and STATISTICS_LEVEL initialization parameters

ADDM analysis of I/O performance partially depends on a single argument, DBIO_EXPECTED, that describes the expected performance of the I/O subsystem. The value of DBIO_EXPECTED is the average time it takes to read a single database block in microseconds. Oracle uses the default value of 10 milliseconds, which is an appropriate value for most modern hard drives. If your hardware is significantly different, such as very old hardware or very fast RAM disks, consider using a different value.

To determine the correct setting for DBIO_EXPECTED parameter:

1. Measure the average read time of a single database block read for your hardware. Note that this measurement is for random I/O, which includes seek time if you use standard hard drives. Typical values for hard drives are between 5000 and 20000 microseconds.
2. Set the value one time for all subsequent ADDM executions. For example, if the measured value if 8000 microseconds, you should execute the following command as SYS user:

```
EXECUTE DBMS_ADVISOR.SET_DEFAULT_TASK_PARAMETER(
                     'ADDM', 'DBIO_EXPECTED', 8000);
```

Diagnosing Database Performance Problems with ADDM

To diagnose database performance problems, first review the ADDM analysis results that are automatically created each time an AWR snapshot is taken. If a different analysis is required (such as a longer analysis period, using a different DBIO_EXPECTED setting, or changing the analysis mode), you can run ADDM manually as described in this section.

ADDM can analyze any two AWR snapshots (on the same database), as long as both snapshots are still stored in the AWR (have not been purged). ADDM can only analyze instances that are started before the beginning snapshot and remain running until the ending snapshot. Additionally, ADDM will not analyze instances that experience significant errors when generating the AWR snapshots. In such cases, ADDM will analyze the largest subset of instances that did not experience these problems.

The primary interface for diagnostic monitoring is Oracle Enterprise Manager. Whenever possible, you should run ADDM using Oracle Enterprise Manager, as described in *Oracle Database 2 Day + Performance Tuning Guide*. If Oracle Enterprise Manager is unavailable, you can run ADDM using the DBMS_ADDM package. In order to run the DBMS_ADDM APIs, the user must be granted the ADVISOR privilege.

This section contains the following topics:

- Running ADDM in Database Mode
- Running ADDM in Instance Mode

- Running ADDM in Partial Mode
- Displaying an ADDM Report

> **See Also:** *Oracle Database PL/SQL Packages and Types Reference* for information about the DBMS_ADDM package

Running ADDM in Database Mode

For Oracle RAC configurations, you can run ADDM in Database mode to analyze all instances of the databases. For single-instance configurations, you can still run ADDM in Database mode; ADDM will simply behave as if running in Instance mode.

To run ADDM in Database mode, use the DBMS_ADDM.ANALYZE_DB procedure:

```
BEGIN
DBMS_ADDM.ANALYZE_DB (
   task_name           IN OUT VARCHAR2,
   begin_snapshot      IN     NUMBER,
   end_snapshot        IN     NUMBER,
   db_id               IN     NUMBER := NULL);
END;
/
```

The task_name parameter specifies the name of the analysis task that will be created. The begin_snapshot parameter specifies the snapshot number of the beginning snapshot in the analysis period. The end_snapshot parameter specifies the snapshot number of the ending snapshot in the analysis period. The db_id parameter specifies the database identifier of the database that will be analyzed. If unspecified, this parameter defaults to the database identifier of the database to which you are currently connected.

The following example creates an ADDM task in database analysis mode, and executes it to diagnose the performance of the entire database during the time period defined by snapshots 137 and 145:

```
VAR tname VARCHAR2(30);
BEGIN
  :tname := 'ADDM for 7PM to 9PM';
  DBMS_ADDM.ANALYZE_DB(:tname, 137, 145);
END;
/
```

Running ADDM in Instance Mode

To analyze a particular instance of the database, you can run ADDM in Instance mode. To run ADDM in Instance mode, use the DBMS_ADDM.ANALYZE_INST procedure:

```
BEGIN
DBMS_ADDM.ANALYZE_INST (
   task_name           IN OUT VARCHAR2,
   begin_snapshot      IN     NUMBER,
   end_snapshot        IN     NUMBER,
   instance_number     IN     NUMBER := NULL,
   db_id               IN     NUMBER := NULL);
END;
/
```

The task_name parameter specifies the name of the analysis task that will be created. The begin_snapshot parameter specifies the snapshot number of the beginning

snapshot in the analysis period. The end_snapshot parameter specifies the snapshot number of the ending snapshot in the analysis period. The instance_number parameter specifies the instance number of the instance that will be analyzed. If unspecified, this parameter defaults to the instance number of the instance to which you are currently connected. The db_id parameter specifies the database identifier of the database that will be analyzed. If unspecified, this parameter defaults to the database identifier of the database to which you are currently connected.

The following example creates an ADDM task in instance analysis mode, and executes it to diagnose the performance of instance number 1 during the time period defined by snapshots 137 and 145:

```
VAR tname VARCHAR2(30);
BEGIN
  :tname := 'my ADDM for 7PM to 9PM';
  DBMS_ADDM.ANALYZE_INST(:tname, 137, 145, 1);
END;
/
```

Running ADDM in Partial Mode

To analyze a subset of all database instances, you can run ADDM in Partial mode. To run ADDM in Partial mode, use the DBMS_ADDM.ANALYZE_PARTIAL procedure:

```
BEGIN
DBMS_ADDM.ANALYZE_PARTIAL (
   task_name           IN OUT VARCHAR2,
   instance_numbers    IN     VARCHAR2,
   begin_snapshot      IN     NUMBER,
   end_snapshot        IN     NUMBER,
   db_id               IN     NUMBER := NULL);
END;
/
```

The task_name parameter specifies the name of the analysis task that will be created. The instance_numbers parameter specifies a comma-delimited list of instance numbers of instances that will be analyzed. The begin_snapshot parameter specifies the snapshot number of the beginning snapshot in the analysis period. The end_snapshot parameter specifies the snapshot number of the ending snapshot in the analysis period. The db_id parameter specifies the database identifier of the database that will be analyzed. If unspecified, this parameter defaults to the database identifier of the database to which you are currently connected.

The following example creates an ADDM task in partial analysis mode, and executes it to diagnose the performance of instance numbers 1, 2, and 4, during the time period defined by snapshots 137 and 145:

```
VAR tname VARCHAR2(30);
BEGIN
  :tname := 'my ADDM for 7PM to 9PM';
  DBMS_ADDM.ANALYZE_PARTIAL(:tname, '1,2,4', 137, 145);
END;
/
```

Displaying an ADDM Report

To display a text report of an executed ADDM task, use the DBMS_ADDM.GET_REPORT function:

```
DBMS_ADDM.GET_REPORT (
   task_name           IN VARCHAR2
  RETURN CLOB);
```

The following example displays a text report of the ADDM task specified by its task name using the `tname` variable:

```
SET LONG 1000000 PAGESIZE 0;
SELECT DBMS_ADDM.GET_REPORT(:tname) FROM DUAL;
```

Note that the return type of a report is a `CLOB`, formatted to fit line size of 80. For information about reviewing the ADDM analysis results in an ADDM report, see "ADDM Analysis Results" on page 6-4.

Views with ADDM Information

Typically, you should view output and information from ADDM using Oracle Enterprise Manager or ADDM reports.

However, you can display ADDM information through the `DBA_ADVISOR` views. This group of views includes:

- `DBA_ADVISOR_FINDINGS`

  This view displays all the findings discovered by all advisors. Each finding is displayed with an associated finding Id, name, and type. For tasks with multiple executions, the name of each task execution associated with each finding is also listed.

- `DBA_ADDM_FINDINGS`

  This view contains a subset of the findings displayed in the related `DBA_ADVISOR_FINDINGS` view. This view only displays the ADDM findings discovered by all advisors. Each ADDM finding is displayed with an associated finding Id, name, and type.

- `DBA_ADVISOR_FINDING_NAMES`

  List of all finding names registered with the advisor framework.

- `DBA_ADVISOR_RECOMMENDATIONS`

  This view displays the results of completed diagnostic tasks with recommendations for the problems identified in each execution. The recommendations should be reviewed in the order of the `RANK` column, as this relays the magnitude of the problem for the recommendation. The `BENEFIT` column displays the benefit to the system you can expect after the recommendation is performed. For tasks with multiple executions, the name of each task execution associated with each advisor task is also listed.

- `DBA_ADVISOR_TASKS`

  This view provides basic information about existing tasks, such as the task Id, task name, and when the task was created. For tasks with multiple executions, the name and type of the last or current execution associated with each advisor task is also listed.

> **See Also:** *Oracle Database Reference* for information on static data dictionary views

7

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. Oracle recommends using automatic memory management to manage the memory on your system. However, you can choose to manually adjust the memory pools on your system, as described in this chapter.

This chapter contains the following sections:

> **See Also:** *Oracle Database Concepts* for information on the memory architecture of an Oracle database

Understanding Memory Allocation Issues

Oracle stores information in memory caches and on disk. Memory access is much faster than disk access. Disk access (physical I/O) take a significant amount of time, compared with memory access, typically in the order of 10 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 of frequently accessed objects to be perform by memory, rather than also requiring disk access.

A performance goal is to reduce the physical I/O overhead as much as possible, either by making it more likely that the required data is in memory, or by making the process of retrieving the required data more efficient.

This section contains the following topics:

- Operating System Memory Use
- Iteration During Configuration

Oracle Memory Caches

The main Oracle memory caches that affect performance are:

- Shared pool
- Large pool
- Java pool
- Buffer cache
- Streams pool size
- Log buffer
- Process-private memory, such as memory used for sorting and hash joins

Automatic Memory Management

Oracle strongly recommends the use of automatic memory management to manage the memory on your system. Automatic memory management enables Oracle Database to automatically manage and tune the instance memory. Automatic memory management can be configured using a target memory size initialization parameter (`MEMORY_TARGET`) and a maximum memory size initialization parameter (`MEMORY_MAX_TARGET`). Oracle Database then tunes to the target memory size, redistributing memory as needed between the system global area (SGA) and the instance program global area (instance PGA). Before setting any memory pool sizes, consider using the automatic memory management feature of Oracle Database. If you need to configure memory allocations, you should also consider using the Memory Advisor for managing memory.

See Also:

- *Oracle Database Administrator's Guide* for information about using automatic memory management
- *Oracle Database 2 Day DBA* for information about using the Memory Advisor

Automatic Shared Memory Management

Automatic Shared Memory Management simplifies the configuration of the SGA. To use Automatic Shared Memory Management, set the `SGA_TARGET` initialization parameter to a nonzero value and set the `STATISTICS_LEVEL` initialization parameter to `TYPICAL` or `ALL`. The value of the `SGA_TARGET` parameter should be set to the amount of memory that you want to dedicate for the SGA. In response to the workload on the system, the automatic SGA management distributes the memory appropriately for the following memory pools:

- Database buffer cache (default pool)
- Shared pool
- Large pool
- Java pool
- Streams pool

If these automatically tuned memory pools had been set to nonzero values, those values are used as minimum levels by Automatic Shared Memory Management. You would set minimum values if an application component needs a minimum amount of memory to function properly.

SGA_TARGET is a dynamic parameter that can be changed by accessing the SGA Size Advisor from the Memory Parameters SGA page in Oracle Enterprise Manager, or by querying the V$SGA_TARGET_ADVICE view and using the ALTER SYSTEM command. SGA_TARGET can be set less than or equal to the value of SGA_MAX_SIZE initialization parameter. Changes in the value of SGA_TARGET automatically resize the automatically tuned memory pools.

See Also:

- *Oracle Database Concepts* for information about the System Global Area (SGA)
- *Oracle Database Administrator's Guide* for information about managing the System Global Area (SGA)

If you dynamically disable SGA_TARGET by setting its value to 0 at instance startup, Automatic Shared Memory Management will be disabled and the current auto-tuned sizes will be used for each memory pool. If necessary, you can manually resize each memory pool using the DB_CACHE_SIZE, SHARED_POOL_SIZE, LARGE_POOL_SIZE, JAVA_POOL_SIZE, and STREAMS_POOL_SIZE initialization parameters. See "Dynamically Changing Cache Sizes" on page 7-3.

The following pools are manually sized components and are not affected by Automatic Shared Memory Management:

- Log buffer
- Other buffer caches (such as KEEP, RECYCLE, and other non-default block size)
- Fixed SGA and other internal allocations

To manually size these memory pools, you need to set the DB_KEEP_CACHE_SIZE, DB_RECYCLE_CACHE_SIZE, DB_nK_CACHE_SIZE, and LOG_BUFFER initialization parameters. The memory allocated to these pools is deducted from the total available for SGA_TARGET when Automatic Shared Memory Management computes the values of the automatically tuned memory pools.

See Also:

- *Oracle Database Administrator's Guide* for information on managing initialization parameters
- *Oracle Streams Concepts and Administration* for information on the STREAMS_POOL_SIZE initialization parameter
- *Oracle Database Java Developer's Guide* for information on Java memory usage

Dynamically Changing Cache Sizes

If the system is not using Automatic Memory Management or Automatic Shared Memory Management, you can choose to dynamically reconfigure the sizes of the shared pool, the large pool, the buffer cache, and the process-private memory. The following sections contain details on sizing of caches:

- Configuring and Using the Buffer Cache

- Configuring and Using the Shared Pool and Large Pool
- Configuring and Using the Redo Log Buffer

The size of these memory caches is configurable using initialization configuration parameters, such as DB_CACHE_ADVICE, JAVA_POOL_SIZE, LARGE_POOL_SIZE, LOG_BUFFER, and SHARED_POOL_SIZE. The values for these parameters are also dynamically configurable using the ALTER SYSTEM statement except for the log buffer pool and process-private memory, which are static after startup.

Memory for the shared pool, large pool, java pool, and buffer cache is allocated in units of granules. The granule size is 4MB if the SGA size is less than 1GB. If the SGA size is greater than 1GB, the granule size changes to 16MB. The granule size is calculated and fixed when the instance starts up. The size does not change during the lifetime of the instance.

The granule size that is currently being used for SGA can be viewed in the view V$SGA_DYNAMIC_COMPONENTS. The same granule size is used for all dynamic components in the SGA.

You can expand the total SGA size to a value equal to the SGA_MAX_SIZE parameter. If the SGA_MAX_SIZE is not set, you can decrease the size of one cache and reallocate that memory to another cache if necessary. SGA_MAX_SIZE defaults to the aggregate setting of all the components.

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 of 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: Your operating system's documentation for information on managing dynamic SGA

Viewing Information About Dynamic Resize Operations

The following views provide information about dynamic resize operations:

- V$MEMORY_CURRENT_RESIZE_OPS displays information about memory resize operations (both automatic and manual) which are currently in progress.
- V$MEMORY_DYNAMIC_COMPONENTS displays information about the current sizes of all dynamically tuned memory components, including the total sizes of the SGA and instance PGA.
- V$MEMORY_RESIZE_OPS displays information about the last 800 completed memory resize operations (both automatic and manual). This does not include in-progress operations.
- V$MEMORY_TARGET_ADVICE displays tuning advice for the MEMORY_TARGET initialization parameter.
- V$SGA_CURRENT_RESIZE_OPS displays information about SGA resize operations that are currently in progress. An operation can be a grow or a shrink of a dynamic SGA component.

- V$SGA_RESIZE_OPS displays information about the last 800 completed SGA resize operations. This does not include any operations currently in progress.
- V$SGA_DYNAMIC_COMPONENTS displays information about the dynamic components in SGA. This view summarizes information based on all completed SGA resize operations since startup.
- V$SGA_DYNAMIC_FREE_MEMORY displays information about the amount of SGA memory available for future dynamic SGA resize operations.

See Also:

- *Oracle Database Concepts* for more information about dynamic SGA
- *Oracle Database Reference* for detailed column information for these views

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 resource requirements. Efficient use of the Oracle memory caches also reduces the load on related resources, such as the latches that protect the caches, the CPU, and the I/O system.

For best performance, you should consider the following:

- The cache should be optimally designed to use the operating system and database resources most 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.

If your application uses Java, you should investigate whether you need to modify the default configuration for the Java pool. See the *Oracle Database Java Developer's Guide* for information on Java memory usage.

Operating System Memory Use

For most operating systems, it is important to consider the following:

Reduce paging

Paging occurs when an operating system transfers memory-resident pages to 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.

Use the operating system utilities to examine the operating system, to identify whether there is a lot of paging on your system. If there is, 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 SGA into main memory

Because the purpose of the 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 the

data is no longer quickly accessible. On most operating systems, the disadvantage of paging significantly outweighs the advantage of a large SGA.

> **Note:** The LOCK_SGA parameter can be used to lock the SGA into physical memory and prevent it from being paged out. The MEMORY_TARGET and MEMORY_MAX_TARGET parameters cannot be used when the LOCK_SGA parameter is enabled.

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 the system.

> **See Also:** Your operating system hardware and software documentation, as well as the Oracle documentation specific to your operating system, 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 the 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 lets you 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 store 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 Advisory Statistics
- Considering Multiple Buffer Pools

Using the Buffer Cache Effectively

To use the buffer cache effectively, SQL statements for the application 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 16, "SQL Tuning Overview" for information on how to do this

Sizing the Buffer Cache

When configuring a new instance, it is impossible to know the correct size for the buffer cache. Typically, a database administrator 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 Advisory Statistics

A number of statistics 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 initialization parameter is set to ON. This view shows the simulated miss rates for a range of potential buffer cache sizes.

Each cache size simulated has its own row in this view, with the predicted physical I/O activity that would take place for that 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 is some overhead associated with this advisory. When the advisory is enabled, there is a small increase in CPU usage, because additional bookkeeping is required.

Oracle uses DBA-based sampling to gather cache advisory statistics. Sampling substantially reduces both CPU and memory overhead associated with bookkeeping. Sampling is not used for a buffer pool if the number of buffers in that buffer pool is small to begin with.

To use V$DB_CACHE_ADVICE, the parameter DB_CACHE_ADVICE should be set to ON, and a representative workload should be running on the instance. Allow the workload to stabilize before querying the V$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 (MB)'
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 212 MB, rather than the current size of 304 MB, the estimated number of physical reads would increase by a factor of 1.74 or 74%. This means it would not be advisable to decrease the cache size to 212MB.

However, increasing the cache size to 334MB would potentially decrease reads by a factor of .93 or 7%. If an additional 30MB memory is available on the host machine and the SGA_MAX_SIZE setting allows the increment, it would be advisable to increase the default buffer cache pool size to 334MB.

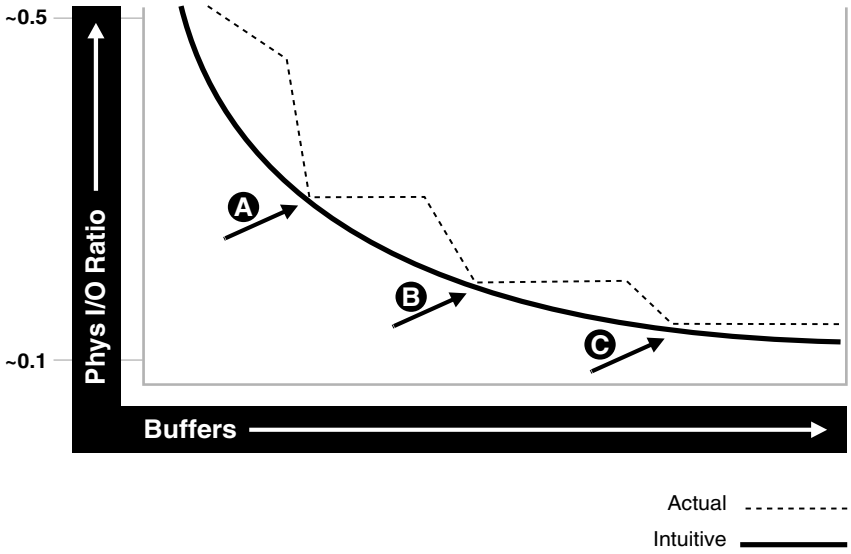
Cache Size (MB)	Buffers	Estd Phys Read Factor	Estd Phys Reads	
30	3,802	18.70	192,317,943	10% of Current Size
60	7,604	12.83	131,949,536	
91	11,406	7.38	75,865,861	
121	15,208	4.97	51,111,658	
152	19,010	3.64	37,460,786	
182	22,812	2.50	25,668,196	
212	26,614	1.74	17,850,847	
243	30,416	1.33	13,720,149	
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.

Note: With Oracle, physical reads do not necessarily indicate disk reads; physical reads may well be satisfied from the file system cache.

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 7–1 on page 7-9, only narrow bands of increments to the cache size may be worthy of consideration.

Figure 7–1 Physical I/O and Buffer Cache Size



Examining Figure 7–1 leads to the following 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.

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`.

The statistics in Table 7–1 are used to calculate the hit ratio.

Table 7–1 Statistics for Calculating the Hit Ratio

Statistic	Description
consistent gets from cache	Number of times a consistent read was requested for a block from the buffer cache.
db block gets from cache	Number of times a CURRENT block was requested from the buffer cache.
physical reads cache	Total number of data blocks read from disk into buffer cache.

Example 7–1 has been simplified by using values selected directly from the `V$SYSSTAT` table, rather than over an interval. It is best to calculate the delta of these statistics over an interval while your application is running, then use them to determine the hit ratio.

> **See Also:** Chapter 6, "Automatic Performance Diagnostics" for more information on collecting statistics over an interval

Example 7–1 Calculating the Buffer Cache Hit Ratio

```
SELECT NAME, VALUE
```

```
  FROM V$SYSSTAT
WHERE NAME IN ('db block gets from cache', 'consistent gets from cache', 'physical
reads cache');
```

Using the values in the output of the query, calculate the hit ratio for the buffer cache with the following formula:

```
1 - (('physical reads cache') / ('consistent gets from cache' + 'db block gets
from cache')
```

> **See Also:** *Oracle Database Reference* for information on the `V$SYSSTAT` view

Interpreting and Using the Buffer Cache Advisory Statistics

There are many factors to examine before considering whether to increase or decrease the buffer cache size. For example, you should examine `V$DB_CACHE_ADVICE` data and the buffer cache hit ratio.

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.

To interpret the buffer cache hit ratio, you should consider the following:

- Repeated scanning of the same large table or index can artificially inflate a poor cache hit ratio. Examine frequently executed SQL statements with a large number of buffer gets, to ensure that the execution plan for such SQL statements is optimal. If possible, avoid repeated scanning of 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.
- Oracle blocks accessed during a long full table scan are put on the tail end of the least recently used (LRU) list and not on the head of the list. Therefore, the blocks are aged out faster than blocks read when performing indexed lookups or small table scans. When interpreting the buffer cache data, poor hit ratios when valid large full table scans are occurring should also be considered.

  > **Note:** Short table scans are scans performed on tables under a certain size threshold. The definition of a small table is the maximum of 2% of the buffer cache and 20, whichever is bigger.

- 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 block 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.

To increase cache size, first set the DB_CACHE_ADVICE initialization 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 operating system to page, then allocate this memory. To increase the amount of memory allocated to the buffer cache, increase the value of the DB_CACHE_SIZE initialization parameter.

If required, resize the buffer pools dynamically, rather than shutting down the instance to perform this change.

Note: When the cache is resized significantly (greater than 20 percent), the old cache advisory value is discarded and the cache advisory is set to the new size. Otherwise, the old cache advisory value is adjusted to the new size by the interpolation of existing values.

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_*n*K_CACHE_SIZE parameter can be used to configure 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: *Oracle Database Reference* and *Oracle Database Administrator's Guide* for more information on using the DB_*n*K_CACHE_SIZE parameters

Reducing Memory Allocated to the Buffer Cache

If the cache hit ratio is high, then the cache is probably large enough to hold the most frequently accessed data. Check V$DB_CACHE_ADVICE data to see whether decreasing the 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 an application's buffer pool might benefit from configuring multiple buffer pools.

With segments that have atypical access patterns, store blocks from those segments in two different buffer pools: the KEEP pool and the RECYCLE pool. A segment's access pattern may be atypical if it is constantly accessed (that is, hot) or infrequently accessed (for example, a large segment accessed by a batch job only once a 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:

- 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 can be considered very large. Random reads to 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. However, the problem affects 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, find out whether the index is selective. If not, tune the SQL statement to use a more selective index.
2. If the SQL statement is tuned, 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, you can 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 application requirements for that instance.

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:** *Oracle Database SQL Reference* for information on specifying BUFFER_POOL in the STORAGE clause

Buffer Pool Data in V$DB_CACHE_ADVICE

V$DB_CACHE_ADVICE can be used to size all pools configured on an instance. Make the initial cache size estimate, run the representative workload, then simply query the V$DB_CACHE_ADVICE view for the pool you want to use.

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$SYSSTAT reflects the logical and physical reads for all buffer pools within one set of statistics. To determine the hit ratio for the buffer pools individually, query the V$BUFFER_POOL_STATISTICS view. This view maintains statistics for each pool on the number of logical reads and writes.

The buffer pool hit ratio can be determined using the following formula:

```
1 - (physical_reads/(db_block_gets + consistent_gets))
```

The ratio can be calculated with 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;
```

> **See Also:** *Oracle Database Reference* for information on the V$BUFFER_POOL_STATISTICS view

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. To determine which segments have many buffers in the pool, you can use one of the two methods described in this section. You can either look at the buffer cache usage pattern for all segments (Method 1) or examine the usage pattern of a specific segment, (Method 2).

Method 1

The following query counts the number of blocks for all segments that reside in the buffer cache at that point in time. Depending on buffer cache size, this might 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(*) NUMBER_OF_BLOCKS
     FROM DBA_OBJECTS o, V$BH bh
    WHERE o.DATA_OBJECT_ID = bh.OBJD
      AND o.OWNER         != 'SYS'
    GROUP BY o.OBJECT_NAME
    ORDER BY COUNT(*);

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 query:

   ```
   SELECT DATA_OBJECT_ID, OBJECT_TYPE
     FROM DBA_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.

2. Find the number of buffers in the buffer cache for `SEGMENT_NAME`:

   ```
   SELECT COUNT(*) BUFFERS
     FROM V$BH
    WHERE OBJD = data_object_id_value;
   ```

   where *`data_object_id_value`* 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 store the 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_CACHE_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 that 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 7-13.

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 more recent values for physical reads, block gets, and consistent gets from the older values, and use the results to compute the hit ratio.

A buffer pool hit ratio of 100% 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 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). Consequently, 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_CACHE_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.

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, result 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 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

The main components of the shared pool are the library cache, the dictionary cache, and, depending on your configuration, the result 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. The result cache stores the results of queries and PL/SQL function results. 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.

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.

A number of features make large memory allocations in the shared pool: for example, 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:** "Using the Large Pool" on page 7-32 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 occur 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:** "Configuring the Reserved Pool" on page 7-36 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 stores 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 code. 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 "SQL Sharing Criteria" on page 7-18 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. See "SQL Execution Efficiency" on page 2-13 for a discussion of hard and soft parsing.

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 performs the following steps for the comparison:

1. The text of the statement issued is compared to existing statements in the shared pool.
2. 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.
3. 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 rule is when the parameter `CURSOR_SHARING` has been set to `SIMILAR` or `FORCE`. Similar statements can 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:** *Oracle Database Reference* for more information on the `CURSOR_SHARING` parameter

4. The objects referenced in the issued statement are compared to the referenced objects of all existing statements in the shared pool to ensure that they are identical.

   References to schema objects in the SQL statements or PL/SQL blocks must resolve to the same object in the same schema. For example, if two users each issue the following SQL statement:

   ```
   SELECT * FROM employees;
   ```

   and they each have their own `employees` table, then this statement is not considered identical, because the statement references different tables for each user.

5. 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.

6. The session's environment must be identical. For example, SQL statements must be optimized using the same optimization goal.

Result Cache Concepts

Systems with large amounts of memory can take advantage of the result cache to improve response times of repetitive queries.

The result cache stores the results of SQL queries and PL/SQL functions in an area called `Result Cache Memory` in the shared pool. When these queries and functions are executed repeatedly, the results are retrieved directly from the cache memory. This results in a faster response time. The cached results stored become invalid when data in the dependent database objects is modified. The use of the result cache is database-wide decision. Result cache itself is instance specific and can be sized differently on different instances. To disable the result cache in a cluster, you must explicitly set the `RESULT_CACHE_MAX_SIZE` initialization parameter to 0 during every instance startup.

The `Result Cache Memory` pool consists of the `SQL Query Result Cache`, which stores the results of SQL queries, and the `PL/SQL Function Result Cache`, which stores the values returned by PL/SQL functions. For details of using the result cache in PL/SQL functions, refer to *Using the Cross-Session PL/SQL Function Result Cache* in the *Oracle Database PL/SQL User's Guide and Reference*.

SQL Query Result Cache Concepts

SQL query results can be cached for reuse. For a read-consistent snapshot to be reusable, one of the following must be true:

- The read-consistent snapshot used to build the result retrieves the most current committed state of the data.
- The query points to an explicit point in time using flashback query.

The use of the SQL query result cache can be controlled by setting the `RESULT_CACHE_MODE` initialization parameter.

RESULT_CACHE_MODE

The `RESULT_CACHE_MODE` initialization parameter determines the SQL query result cache behavior. The possible initialization parameter values are `MANUAL` and `FORCE`.

When set to `MANUAL`, you must specify, by using the `result_cache` hint, that a particular result is using the cache. If the result is not available in the cache, then the query will be executed and the result will be stored in the cache. Subsequent executions of the exact same statement, including the result cache hint, will be served out of the cache.

When set to `FORCE`, all results use the cache, if possible. You can use the `no_result_cache` hint to bypass the cache when using the `FORCE` mode.

A least-recently used algorithm is used to age out cached results. Query results that are bigger than the available space in the result cache will not be cached.

The ResultCache Operator

To store the result of a query in the result cache when RESULT_CACHE_MODE is set to MANUAL, you must include the result_cache hint in your query. For example:

```
select /*+ result_cache */ deptno, avg(sal)
from emp
group by deptno;
```

This hint introduces the ResultCache operator into the execution plan for the query. When you execute the query, the ResultCache operator will look up the result cache memory to check if the result for the query already exists in the cache. If it exists, then the result is retrieved directly out of the cache. If it does not yet exist in the cache, then the query is executed. The result is returned as output, and is also stored in the result cache memory.

If the RESULT_CACHE_MODE is set to FORCE, and you do not wish to include the result of a query to the result cache, then you must use the no_result_cache hint in your query.

Restrictions on Using the SQL Query Result Cache

You cannot cache results when you use the following database objects or functions in your SQL query:

- Dictionary and temporary tables
- Sequence CURRVAL and NEXTVAL pseudo columns
- SQL functions current_date, current_timestamp, local_timestamp, userenv/sys_context (with non-constant variables), sys_guid, sysdate, and sys_timestamp
- Non-deterministic PL/SQL functions

Cached results are parameterized with the parameter values if any of the following constructs are used in the query:

- Bind variables.
- The following SQL functions: dbtimezone, sessiontimezone, userenv/sys_context (with constant variables), uid, and user.
- NLS parameters.

Parameterized cache results can be reused if the query is equivalent and the parameter values are the same.

A query result based on a read-consistent snapshot of data that is older than the latest committed version of the data will not be cached. If any of the tables used to build a cached result has been modified in an ongoing transaction in the current session then the result is never cached.

Adding the RESULT_CACHE hint to inline views disables optimizations between the outer query and the inline view in order to maximize the reusability of the cached result. Subqueries cannot be cached.

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 for a 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.

In a data warehousing environment, the SQL query result cache also enables you to optimize the use of the shared pool.

> **See Also:** *Oracle Database 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 items with application developers and agree on strategies to ensure that the shared pool is used effectively:

- Shared Cursors
- Single-User Logon and Qualified Table Reference
- Use of PL/SQL
- Avoid Performing DDL
- Cache Sequence Numbers
- Cursor Access and Management
- Use of Result Cache

Efficient use of the shared pool in high-concurrency OLTP systems significantly reduces the probability of parse-related application scalability issues.

Shared Cursors

Reuse of shared SQL for multiple users running the same application, avoids hard parsing. Soft parses provide a significant reduction in the use of resources such as the 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 "CURSOR_SHARING for Existing Applications" on page 7-38.

- 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 for application developers:
  - 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 use the same shared PL/SQL area automatically. Because stored procedures are stored in a parsed form, their use reduces runtime parsing.
- For SQL statements which are identical but are not being shared, you can query `V$SQL_SHARED_CURSOR` to determine why the cursors are not shared. This would include optimizer settings and bind variable mismatches.

Single-User Logon and Qualified Table Reference

Large OLTP systems where users log in to the database as their own user ID can benefit from explicitly qualifying the segment owner, rather than 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 through a single user ID, rather than individual user IDs. 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 many 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 the dictionary cache load considerably.

> **Note:** Oracle encourages the use of definer's rights packages to overcome scalability issues. The benefits of reduced dictionary cache load are not as obvious with invoker's rights packages.

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 space 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: *Oracle Database 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), 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, 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 sections introduce the methods used for some Oracle tools.

See Also:

- The tool-specific documentation for more information about each tool
- *Oracle Database Concepts* for more information on cursors shared SQL

Reducing Parse Calls with OCI When using Oracle Call Interface (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 for 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:** *Oracle Database JDBC Developer's Guide and Reference* 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.

Use of Result Cache

OLTP applications can benefit significantly from the use of the result cache. The benefits highly depend on the application. Consider the use of the PL/SQL function result cache and the SQL query result cache when evaluating whether your application can benefit from the result cache.

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-configured or over-configured.

For most OLTP applications, shared pool size is an important factor in application performance. Shared pool size is less important for applications that issue a very limited number of discrete SQL statements, such as decision support systems (DSS).

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. When examining the statistics after the database has been running, a DBA should check that none of these mistakes are in the workload.

Shared Pool: Library Cache Statistics

When sizing the shared pool, the goal is to ensure that SQL statements that will be executed multiple times are cached in the library cache, 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 will have a value near zero.

The `INVALIDATIONS` column in `V$LIBRARYCACHE` 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 other statistics discussed in this section 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 in more detail in the following section.

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 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:** *Oracle Database Reference* for information about the dynamic performance 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, the cache hit ratio is the following:

```
SUM(PINHITS)/SUM(PINS)
----------------------
            .999466248
```

Note: These queries return data from instance startup, rather than statistics gathered during an interval; interval statistics can better pinpoint the problem.

See Also: Chapter 6, "Automatic Performance Diagnostics" for information on how gather information over an interval

Examining the returned data leads to the following 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 statements 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 free memory is always 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 Advisory Statistics

The amount of memory available for the library cache can drastically affect the parse rate of an Oracle instance. The shared pool advisory statistics provide a database administrator with information about library cache memory, allowing a DBA to predict how changes in the size of the shared pool can affect aging out of objects in the shared pool.

The shared pool advisory statistics track the library cache's use of shared pool memory and predict how the library cache will behave in shared pools of different sizes. Two fixed views provide the information to determine how much memory the library cache is using, how much is currently pinned, how much is on the shared pool's LRU list, as well as how much time might be lost or gained by changing the size of the shared pool.

The following views of the shared pool advisory statistics are available. These views display any data when shared pool advisory is on. These statistics reset when the advisory is turned off.

V$SHARED_POOL_ADVICE This view displays information about estimated parse time in the shared pool for different pool sizes. The sizes range from 10% of the current shared pool size or the amount of pinned library cache memory, whichever is higher, to 200% of the current shared pool size, in equal intervals. The value of the interval depends on the current size of the shared pool.

V$LIBRARY_CACHE_MEMORY This view displays information about memory allocated to library cache memory objects in different namespaces. A memory object is an internal grouping of memory for efficient management. A library cache object may consist of one or more memory objects.

V$JAVA_POOL_ADVICE and V$JAVA_LIBRARY_CACHE_MEMORY These views contain Java pool advisory statistics that track information about library cache memory used for Java and predict how changes in the size of the Java pool can affect the parse rate.

V$JAVA_POOL_ADVICE displays information about estimated parse time in the Java pool for different pool sizes. The sizes range from 10% of the current Java pool size or the amount of pinned Java library cache memory, whichever is higher, to 200% of the current Java pool size, in equal intervals. The value of the interval depends on the current size of the Java pool.

> **See Also:** *Oracle Database Reference* for information about the dynamic performance V$SHARED_POOL_ADVICE, V$LIBRARY_CACHE_MEMORY, V$JAVA_POOL_ADVICE, and V$JAVA_LIBRARY_CACHE_MEMORY view

Shared Pool: Dictionary Cache Statistics

Typically, if the shared pool is adequately sized for the library cache, it will also be adequate for the dictionary cache data.

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$ROWCACHE view contains statistics for a single type of data dictionary item. These statistics reflect all data dictionary activity since the most recent instance startup. The columns in the V$ROWCACHE view that reflect the use and effectiveness of the data dictionary cache are listed in Table 7–2.

Table 7–2 V$ROWCACHE Columns

Column	Description
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 description data.

Table 7–2 (Cont.) V$ROWCACHE Columns

Column	Description
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 query 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 using the following formula; however, 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;
```

Shared Pool: Result Cache Statistics

The DBMS_RESULT_CACHE package provides statistics, information, and operators that enable you to manage memory allocation for the result cache.

You can use the DBMS_RESULT_CACHE package to perform various operations such as bypassing the cache, retrieving statistics on the cache memory usage, and flushing the cache. For example, to view the memory allocation statistics, use the following SQL procedure:

```
SQL> set serveroutput on
SQL> execute dbms_result_cache.memory_report
```

The output of this command will be similar to the following:

```
R e s u l t C a c h e M e m o r y R e p o r t
[Parameters]
Block Size = 1024 bytes
Maximum Cache Size = 950272 bytes (928 blocks)
Maximum Result Size = 47104 bytes (46 blocks)
[Memory]
Total Memory = 46340 bytes [0.048% of the Shared Pool]
... Fixed Memory = 10696 bytes [0.011% of the Shared Pool]
... State Object Pool = 2852 bytes [0.003% of the Shared Pool]
... Cache Memory = 32792 bytes (32 blocks) [0.034% of the Shared Pool]
....... Unused Memory = 30 blocks
....... Used Memory = 2 blocks
.......... Dependencies = 1 blocks

.......... Results = 1 blocks
.............. SQL = 1 blocks

PL/SQL procedure successfully completed.
```

To remove all existing results and clear the cache memory, use the command:

```
SQL>execute dbms_result_cache.flush
```

For detailed information on the DBMS_RESULT_CACHE package, refer to *Oracle Database PL/SQL Packages and Types Reference*.

Accessing Information on the Result Cache

Table 7–3 lists the views that provide information and statistics on the result cache memory. The views show the aggregated statistics for the SQL query result cache and the PL/SQL function result cache:

Table 7–3 RESULT_CACHE Views

View Name	Description
(G)V$_RESULT_CACHE_STATISTICS	Lists the various cache settings and memory usage statistics
(G)V$RESULT_CACHE_MEMORY	Lists all the memory blocks and the corresponding statistics
(G)V$RESULT_CACHE_OBJECTS	Lists all the objects (cached results and dependencies) along with their attributes

Table 7–3 (Cont.) RESULT_CACHE Views

View Name	Description
(G)V$RESULT_CACHE_DEPENDENCY	Lists the dependency details between the cached results and dependencies

Use the following query to monitor the statistics in the V$RESULT_CACHE_STATISTICS view:

```
column name format a20
select name, value from v$result_cache_statistics;
```

The output of this query will be similar to the following:

```
NAME                          VALUE
--------------------      ----------
Block Size (Bytes)             1024
Block Count Maximum           3136
Block Count Current             32
Result Size Maximum (Blocks)   156
Create Count Success             2
Create Count Failure             0
Find Count                       0
Invalidation Count               0
Delete Count Invalid             0
Delete Count Valid               0
```

A system that makes good use of the SQL query result cache should show relatively low values for `Create Count Failure` and `Delete Count Valid`, while showing relatively high values for `Find Count`.

For more details on these views, refer to *Oracle Database Reference*.

Interpreting Shared Pool Statistics

Shared pool statistics indicate adjustments that can be made. The following sections describe some of your choices.

Increasing Memory Allocation

Increasing the amount of memory for the shared pool increases the amount of memory available to the library cache, the dictionary cache, and the result 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.

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 7-21.
- 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.

Allocating Additional Memory to the Result Cache By default, on database startup, Oracle allocates memory to the result cache in the shared pool. The memory size allocated depends on the memory size of the shared pool as well as the memory management system.

When using the MEMORY_TARGET initialization parameter to specify the memory allocation, Oracle allocates 0.25% of memory_target to the result cache.

When you set the size of the shared pool using the SGA_TARGET initialization parameter, Oracle allocates 0.5% of sga_target to the result cache.

If you specify the size of the shared pool using the SHARED_POOL_SIZE initialization parameter, then Oracle allocates 1% of the shared pool size to the result cache.

You can change the memory allocated to the result cache by setting the RESULT_CACHE_MAX_SIZE initialization parameter. The result cache is disabled if you set the value to 0 during database startup. RESULT_CACHE_MAX_SIZE cannot be dynamically changed if the value is set to 0 during database startup in the spfile or the init.ora file.

Note: Oracle will not allocate more than 75% of the shared pool to the result cache.

Use the RESULT_CACHE_MAX_RESULT initialization parameter to specify the maximum percentage of result cache memory that can be used by any single result. The default value is 5%, but you can specify any percent value between 1 and 100.

Use the RESULT_CACHE_REMOTE_EXPIRATION initialization parameter to specify the time (in minutes) for which a result that accesses remote database objects remains valid. The default value is 0. When set to 0, the SQL query result cache is disabled for queries that access remote tables. Note that when you use a non zero value for RESULT_CACHE_REMOTE_EXPIRATION, a DML on the remote database will not invalidate the result cache.

Note: Currently, query result cache statistics are not included in V$SHARED_POOL_ADVICE. Therefore, if you create a large result cache then you must add the cache size to the optimal shared pool size from V$SHARED_POOL_ADVICE.

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.

Using the Large Pool

Unlike the shared pool, the large pool does not have an LRU list. Oracle does not attempt to age objects 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:** *Oracle Database 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:** *Oracle Database Backup and Recovery 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 the shared server-related User Global Area (UGA), rather that using the shared pool. This is because Oracle uses the shared pool to allocate System Global Area (SGA) 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 in, 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 to 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) for each 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:

- *Oracle Database Concepts* for more information about the large pool
- *Oracle 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; 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`. Table 7–4 lists these statistics.

Table 7–4 V$SESSTAT Statistics Reflecting Memory

Statistic	Description
`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`. If you are using a shared server, you can use the following query to decide how much larger to make the shared pool. 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 with a location that 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 size of the memory 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.

If you use a shared server architecture, you can use the result of either of these queries to determine how much larger to make the shared pool. 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 for Each 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 on a user-by-user basis.

> **See Also:** *Oracle Database 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 transaction process (TP) monitor is feasible only with pure transactional models, because locks and uncommitted DMLs cannot be held between calls. A shared server environment offers the following advantages:

- It 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.
- It substantially reduces overall memory usage, even though more SGA is used in shared server mode.

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 CURSOR_SPACE_FOR_TIME 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 CURSOR_SPACE_FOR_TIME 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.

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. To minimize the impact on performance, session cursors can be stored in a session cursor cache. These cursors are those that have been closed by the application and can be reused. This feature can be particularly useful for applications that use 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.

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 5 KB) of memory. (The default minimum reserved pool allocation is 4,400 bytes.)

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 the latch resource for detectable periods of time, causing minor disruption to other concurrent attempts at memory allocation.

Hence, Oracle internally reserves a small memory area in the shared pool that can be used if the shared pool does not have enough space. This reserved pool makes allocation of large chunks more efficient.

By default, Oracle configures a small reserved pool. This memory can be used for operations such as PL/SQL and trigger compilation or for temporary space while loading Java objects. After the memory allocated from the reserved pool is freed, it returns to the reserved pool.

You probably will not 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, Oracle attempts to allocate space in the shared pool in the following order:

1. From the unreserved part of the shared pool.
2. From the reserved pool. If there is not enough space in the unreserved part of the shared pool, then Oracle checks whether the reserved pool has enough space.
3. From memory. 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` to more than half of `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 the value of REQUEST_MISSES equal zero. 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 these target values, 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:** *Oracle 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 parameters 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_MISSES is zero or not increasing
- FREE_MEMORY is greater than or equal to 50% of SHARED_POOL_RESERVED_SIZE minimum

If either of these conditions 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 is greater than zero 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.

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: *Oracle Database PL/SQL Packages and Types Reference* for specific information on using DBMS_SHARED_POOL procedures

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 7-18 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 "Enabling Query Optimizer Features" on page 11-5.

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 both of the following questions are true:

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.

Maintaining Connections

Large OLTP applications with middle tiers should maintain connections, rather than connecting and disconnecting for each database request. Maintaining persistent connections saves CPU resources and database resources, such as latches.

See Also: "Operating System Statistics" on page 5-4 for a description of important operating system statistics

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 (too small a 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 can temporarily mask the effect of slower disks in this situation. Alternatively, you can do one of the following:

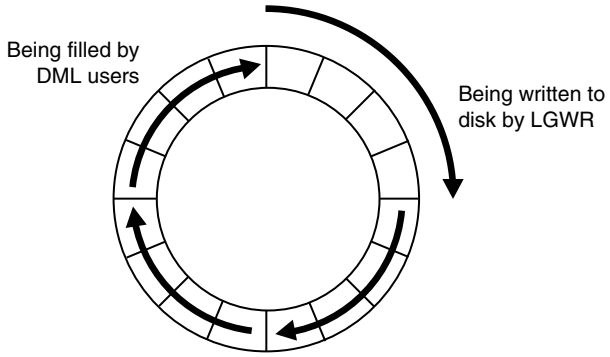
- Improve the checkpointing or archiving process
- Improve the performance of log writer (perhaps by moving all online logs to fast raw devices)

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 7–2 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 the default value, which is:

```
MAX(0.5M, (128K * number of cpus))
```

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 through 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.

PGA Memory Management

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 memory region of the server process executing that cursor.

> **Note:** Part of the runtime area can be located in the SGA when using shared servers.

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, such as `ORDER BY`, `GROUP BY`, `ROLLUP`, and 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 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.

Automatic PGA memory management simplifies and improves the way PGA memory is allocated. By default, PGA memory management is enabled. In this mode, Oracle dynamically adjusts the size of the portion of the PGA memory dedicated to work areas, based on 20% of the SGA memory size. The minimum value is 10MB.

Note: For backward compatibility, automatic PGA memory management can be disabled by setting the value of the PGA_AGGREGATE_TARGET initialization parameter to 0. When automatic PGA memory management is disabled, the maximum size of a work area can be sized with the associated _AREA_SIZE parameter, such as the SORT_AREA_SIZE initialization parameter.

See *Oracle Database Reference* for information on the PGA_AGGREGATE_TARGET, SORT_AREA_SIZE, HASH_AREA_SIZE, BITMAP_MERGE_AREA_SIZE and CREATE_BITMAP_AREA_SIZE initialization parameters.

Configuring Automatic PGA Memory

When running under the automatic PGA memory management mode, sizing of work areas for all sessions becomes automatic and 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 active work areas in the instance is automatically derived from the PGA_AGGREGATE_TARGET initialization parameter. This amount is set to the value of PGA_AGGREGATE_TARGET minus the amount of 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.

Under automatic PGA memory management mode, the main goal of Oracle is to honor the PGA_AGGREGATE_TARGET limit set by the DBA, by controlling dynamically the amount of PGA memory allotted to SQL work areas. At the same time, Oracle tries to maximize the performance of all the memory-intensive SQL operations, by maximizing the number of work areas that are using an optimal amount of PGA memory (cache memory). The rest of the work areas are executed in one-pass mode, unless the PGA memory limit set by the DBA with the parameter PGA_AGGREGATE_TARGET is so low that multi-pass execution is required to reduce even more the consumption of PGA memory and honor the PGA target limit.

When configuring a brand new instance, it is hard to know precisely the appropriate setting for PGA_AGGREGATE_TARGET. You can determine this setting in three stages:

1. Make a first estimate for PGA_AGGREGATE_TARGET, based on a rule of thumb. By default, Oracle uses 20% of the SGA size. However, this initial setting may be too low for a large DSS system.
2. Run a representative workload on the instance and monitor performance, using PGA statistics collected by Oracle, to see whether the maximum PGA size is under-configured or over-configured.
3. Tune PGA_AGGREGATE_TARGET, using Oracle PGA advice statistics.

See Also: *Oracle Database Reference* for information on the PGA_AGGREGATE_TARGET initialization parameter

The following sections explain this in detail:

- Setting PGA_AGGREGATE_TARGET Initially
- Monitoring the Performance of the Automatic PGA Memory Management
- Tuning PGA_AGGREGATE_TARGET

Setting PGA_AGGREGATE_TARGET Initially

The value of the PGA_AGGREGATE_TARGET initialization parameter (for example 100000 KB, 2500 MB, or 50 GB) should be set based on the total amount of memory available for the Oracle instance. This value can then be tuned and dynamically modified at the instance level. Example 7–2 illustrates a typical situation.

Example 7–2 Initial Setting of PGA_AGGREGATE_TARGET

Assume that an Oracle instance is configured to run on a system with 4 GB 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. You might decide to dedicate only 80% (3.2 GB) of the available memory to the Oracle instance.

You must then divide the resulting memory between the SGA and the PGA.

- For OLTP systems, the PGA memory typically accounts for a small fraction of the total memory available (for example, 20%), leaving 80% for the SGA.
- For DSS systems running large, memory-intensive queries, PGA memory can typically use up to 70% of that total (up to 2.2 GB in this example).

Good initial values for the parameter PGA_AGGREGATE_TARGET might be:

- For OLTP: PGA_AGGREGATE_TARGET = (*total_mem* * 80%) * 20%
- For DSS: PGA_AGGREGATE_TARGET = (*total_mem* * 80%) * 50%

  where *total_mem* is the total amount of physical memory available on the system.

In this example, with a value of *total_mem* equal to 4 GB, you can initially set PGA_AGGREGATE_TARGET to 1600 MB for a DSS system and to 655 MB for an OLTP system.

Monitoring the Performance of the Automatic PGA Memory Management

Before starting the tuning process, you need to know how to monitor and interpret the key statistics collected by Oracle to help in assessing the performance of the automatic PGA memory management component. Several dynamic performance views are available for this purpose:

- V$PGASTAT
- V$PROCESS
- V$PROCESS_MEMORY
- V$SQL_WORKAREA_HISTOGRAM
- V$SQL_WORKAREA_ACTIVE
- V$SQL_WORKAREA

V$PGASTAT This view gives instance-level statistics on the PGA memory usage and the automatic PGA memory manager. For example:

```
SELECT * FROM V$PGASTAT;
```

The output of this query might look like the following:

```
NAME                                                     VALUE UNIT
-------------------------------------------------------- ---------- ------------
aggregate PGA target parameter                         41156608 bytes
aggregate PGA auto target                              21823488 bytes
```

```
global memory bound                              2057216 bytes
total PGA inuse                                 16899072 bytes
total PGA allocated                             35014656 bytes
maximum PGA allocated                          136795136 bytes
total freeable PGA memory                         524288 bytes
PGA memory freed back to OS                   1713242112 bytes
total PGA used for auto workareas                    0 bytes
maximum PGA used for auto workareas              2383872 bytes
total PGA used for manual workareas                  0 bytes
maximum PGA used for manual workareas            8470528 bytes
over allocation count                                291
bytes processed                               2124600320 bytes
extra bytes read/written                        39949312 bytes
cache hit percentage                               98.15 percent
```

The main statistics displayed in V$PGASTAT are as follows:

- aggregate PGA target parameter: This is the current value of the initialization parameter PGA_AGGREGATE_TARGET. The default value is 20% of the SGA size. If you set this parameter to 0, automatic management of the PGA memory is disabled.
- aggregate PGA auto target: This gives the amount of PGA memory Oracle can use for work areas running in automatic mode. This amount is dynamically derived from the value of the parameter PGA_AGGREGATE_TARGET and the current work area workload. Hence, it is continuously adjusted by Oracle. If this value is small compared to the value of 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 sort work areas. You must ensure that enough PGA memory is left for work areas running in automatic mode.
- 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 the current state of the work area workload. The global memory bound generally decreases when the number of active work areas is increasing in the system. As a rule of thumb, the value of the global bound should not decrease to less than 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 less than the value of PGA_AGGREGATE_TARGET. However, it is possible for the PGA allocated to exceed that value by a small percentage and for a short period of time, when the work area workload is increasing very rapidly or when the initialization parameter PGA_AGGREGATE_TARGET is set to a too small value.
- total freeable PGA memory: This indicates how much allocated PGA memory which can be freed.
- total PGA used for auto workareas: This indicates how much PGA memory is currently consumed by work areas running under automatic memory management mode. This number can be used to determine how much memory is consumed by other consumers of the PGA memory (for example, PL/SQL or Java):

  ```
  PGA other = total PGA allocated - total PGA used for auto workareas
  ```

- over allocation count: This statistic is cumulative from instance start-up. Over-allocating PGA memory can happen if the value of PGA_AGGREGATE_TARGET is too small to accommodate the PGA other

component in the previous equation plus the minimum memory required to execute the work area workload. When this happens, Oracle cannot honor the initialization parameter PGA_AGGREGATE_TARGET, and extra PGA memory needs to be allocated. If over-allocation occurs, you should increase the value of PGA_AGGREGATE_TARGET using the information provided by the advice view V$PGA_TARGET_ADVICE.

- total bytes processed: This is the number of bytes processed by memory-intensive SQL operators since instance start-up. For example, the number of byte processed is the input size for a sort operation. This number is used to compute the cache hit percentage metric.
- extra bytes read/written: When a work area cannot run optimally, one or more extra passes is performed over the input data. extra bytes read/written represents the number of bytes processed during these extra passes since instance start-up. This number is also used to compute the cache hit percentage. Ideally, it should be small compared to total bytes processed.
- cache hit percentage: This metric is computed by Oracle to reflect the performance of the PGA memory component. It is cumulative from instance start-up. A value of 100% means that all work areas executed by the system since instance start-up have used an optimal amount of PGA memory. This is, of course, ideal but rarely happens except maybe for pure OLTP systems. In reality, some work areas run one-pass or even multi-pass, depending on the overall size of the PGA memory. When a work area cannot run optimally, one or more extra passes is performed over the input data. This reduces the cache hit percentage in proportion to the size of the input data and the number of extra passes performed. Example 7–3 shows how cache hit percentage is affected by extra passes.

Example 7–3 Calculating Cache Hit Percentage

Consider a simple example: Four sort operations have been executed, three were small (1 MB of input data) and one was bigger (100 MB of input data). The total number of bytes processed (BP) by the four operations is 103 MB. If one of the small sorts runs one-pass, an extra pass over 1 MB of input data is performed. This 1 MB value is the number of extra bytes read/written, or EBP. The cache hit percentage is calculated by the following formula:

```
BP x 100 / (BP + EBP)
```

The cache hit percentage in this case is 99.03%, almost 100%. This value reflects the fact that only one of the small sorts had to perform an extra pass while all other sorts were able to run optimally. Hence, the cache hit percentage is almost 100%, because this extra pass over 1 MB represents a tiny overhead. On the other hand, if the big sort is the one to run one-pass, then EBP is 100 MB instead of 1 MB, and the cache hit percentage falls to 50.73%, because the extra pass has a much bigger impact.

V$PROCESS This view has one row for each Oracle process connected to the instance. The columns PGA_USED_MEM, PGA_ALLOC_MEM, PGA_FREEABLE_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_FREEABLE_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_FREEABLE_MEM PGA_MAX_MEM
```

```
------------------------------------ ------------ ------------ --------------- -----------
PSEUDO                                          0            0               0           0
oracle@dlsun1690 (PMON)                    314540       685860               0      685860
oracle@dlsun1690 (MMAN)                    313992       685860               0      685860
oracle@dlsun1690 (DBW0)                    696720      1063112               0     1063112
oracle@dlsun1690 (LGWR)                  10835108     22967940               0    22967940
oracle@dlsun1690 (CKPT)                    352716       710376               0      710376
oracle@dlsun1690 (SMON)                    541508       948004               0     1603364
oracle@dlsun1690 (RECO)                    323688       685860               0      816932
oracle@dlsun1690 (q001)                    233508       585128               0      585128
oracle@dlsun1690 (QMNC)                    314332       685860               0      685860
oracle@dlsun1690 (MMON)                    885756      1996548          393216     1996548
oracle@dlsun1690 (MMNL)                    315068       685860               0      685860
oracle@dlsun1690 (q000)                    330872       716200           65536      716200
oracle@dlsun1690 (TNS V1-V3)               635768       928024               0     1255704
oracle@dlsun1690 (CJQ0)                    533476      1013540               0     1144612
oracle@dlsun1690 (TNS V1-V3)               430648       812108               0      812108
```

V$PROCESS_MEMORY This view displays dynamic PGA memory usage by named component categories for each Oracle process. This view will contain up to six rows for each Oracle process, one row for:

- Each named component category:
  - Java
  - PL/SQL
  - OLAP
  - SQL
- Freeable - memory that has been allocated to the process by the operating system, but not to a specific category.
- Other - memory that has been allocated to a category, but not to one of the named categories.

The columns `CATEGORY`, `ALLOCATED`, `USED`, and `MAX_ALLOCATED` can be used to dynamically monitor the PGA memory usage of Oracle processes for each of the six categories.

> **See Also:** *Oracle Database Reference* for more information on the `V$PROCESS_MEMORY` view.

V$SQL_WORKAREA_HISTOGRAM This view shows the number of work areas executed with optimal memory size, one-pass memory size, and multi-pass memory size since instance start-up. Statistics in this view are subdivided into buckets that are defined by the optimal memory requirement of the work area. Each bucket is identified by a range of optimal memory requirements specified by the values of the columns `LOW_OPTIMAL_SIZE` and `HIGH_OPTIMAL_SIZE`.

Example 7–3 and Example 7–4 show two ways of using `V$SQL_WORKAREA_HISTOGRAM`.

Example 7–4 Querying V$SQL_WORKAREA_HISTOGRAM: Non-empty Buckets

Consider a sort operation that requires 3 MB of memory to run optimally (cached). Statistics about the work area used by this sort are placed in the bucket defined by `LOW_OPTIMAL_SIZE = 2097152` (2 MB) and `HIGH_OPTIMAL_SIZE = 4194303` (4 MB minus 1 byte), because 3 MB falls within that range of optimal sizes. Statistics

are segmented by work area size, because the performance impact of running a work area in optimal, one-pass or multi-pass mode depends mainly on the size of that work area.

The following query shows statistics for all non-empty buckets. Empty buckets are removed with the predicate `where total_execution != 0`.

```
SELECT LOW_OPTIMAL_SIZE/1024 low_kb,
       (HIGH_OPTIMAL_SIZE+1)/1024 high_kb,
       OPTIMAL_EXECUTIONS, ONEPASS_EXECUTIONS, MULTIPASSES_EXECUTIONS
  FROM V$SQL_WORKAREA_HISTOGRAM
 WHERE TOTAL_EXECUTIONS != 0;
```

The result of the query might look like the following:

```
LOW_KB HIGH_KB OPTIMAL_EXECUTIONS ONEPASS_EXECUTIONS MULTIPASSES_EXECUTIONS
------ ------- ------------------ ------------------ ----------------------
     8      16             156255                  0                      0
    16      32                150                  0                      0
    32      64                 89                  0                      0
    64     128                 13                  0                      0
   128     256                 60                  0                      0
   256     512                  8                  0                      0
   512    1024                657                  0                      0
  1024    2048                551                 16                      0
  2048    4096                538                 26                      0
  4096    8192                243                 28                      0
  8192   16384                137                 35                      0
 16384   32768                 45                107                      0
 32768   65536                  0                153                      0
 65536  131072                  0                 73                      0
131072  262144                  0                 44                      0
262144  524288                  0                 22                      0
```

The query result shows that, in the 1024 KB to 2048 KB bucket, 551 work areas used an optimal amount of memory, while 16 ran in one-pass mode and none ran in multi-pass mode. It also shows that all work areas under 1 MB were able to run in optimal mode.

Example 7–5 Querying V$SQL_WORKAREA_HISTOGRAM: Percent Optimal

You can also use `V$SQL_WORKAREA_HISTOGRAM` to find the percentage of times work areas were executed in optimal, one-pass, or multi-pass mode since start-up. This query only considers work areas of a certain size, with an optimal memory requirement of at least 64 KB.

```
SELECT optimal_count, round(optimal_count*100/total, 2) optimal_perc,
       onepass_count, round(onepass_count*100/total, 2) onepass_perc,
       multipass_count, round(multipass_count*100/total, 2) multipass_perc
FROM
 (SELECT decode(sum(total_executions), 0, 1, sum(total_executions)) total,
         sum(OPTIMAL_EXECUTIONS) optimal_count,
         sum(ONEPASS_EXECUTIONS) onepass_count,
         sum(MULTIPASSES_EXECUTIONS) multipass_count
    FROM v$sql_workarea_histogram
   WHERE low_optimal_size > 64*1024);
```

The output of this query might look like the following:

```
OPTIMAL_COUNT OPTIMAL_PERC ONEPASS_COUNT ONEPASS_PERC MULTIPASS_COUNT MULTIPASS_PERC
------------- ------------ ------------- ------------ --------------- --------------
         2239        81.63           504        18.37               0              0
```

This result shows that 81.63% of these work areas have been able to run using an optimal amount of memory. The rest (18.37%) ran one-pass. None of them ran multi-pass. Such behavior is preferable, for the following reasons:

- Multi-pass mode can severely degrade performance. A high number of multi-pass work areas has an exponentially adverse effect on the response time of its associated SQL operator.
- Running one-pass does not require a large amount of memory; only 22 MB is required to sort 1 GB of data in one-pass mode.

V$SQL_WORKAREA_ACTIVE This view can be used to display the work areas that are active (or executing) in the instance. Small active sorts (under 64 KB) are excluded from the view. Use this view to precisely monitor the size of all active work areas and to determine if these active work areas spill to a temporary segment. Example 7–6 shows a typical query of this view:

Example 7–6 Querying V$SQL_WORKAREA_ACTIVE

```
SELECT to_number(decode(SID, 65535, NULL, SID)) sid,
       operation_type OPERATION,
       trunc(EXPECTED_SIZE/1024) ESIZE,
       trunc(ACTUAL_MEM_USED/1024) MEM,
       trunc(MAX_MEM_USED/1024) "MAX MEM",
       NUMBER_PASSES PASS,
       trunc(TEMPSEG_SIZE/1024) TSIZE
  FROM V$SQL_WORKAREA_ACTIVE
 ORDER BY 1,2;

The output of this query might look like the following:
SID         OPERATION     ESIZE       MEM   MAX MEM  PASS   TSIZE
--- ----------------- --------- --------- --------- ----- -------
  8   GROUP BY (SORT)       315       280       904     0
  8         HASH-JOIN      2995      2377      2430     1   20000
  9   GROUP BY (SORT)     34300     22688     22688     0
 11         HASH-JOIN     18044     54482     54482     0
 12         HASH-JOIN     18044     11406     21406     1  120000
```

This output shows that session 12 (column `SID`) is running a hash-join having its work area running in one-pass mode (`PASS` column). This work area is currently using 11406 KB of memory (`MEM` column) and has used, in the past, up to 21406 KB of PGA memory (`MAX MEM` column). It has also spilled to a temporary segment of size 120000 KB. Finally, the column `ESIZE` indicates the maximum amount of memory that the PGA memory manager expects this hash-join to use. This maximum is dynamically computed by the PGA memory manager according to workload.

When a work area is deallocated—that is, when the execution of its associated SQL operator is complete—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.

Example 7–7 shows three typical queries on the `V$SQL_WORKAREA` dynamic view:

Example 7–7 Querying V$SQL_WORKAREA

The following query 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 query finds the cursors with one or more work areas that have been executed in one or even multiple passes:

```
col sql_text format A80 wrap
SELECT sql_text, sum(ONEPASS_EXECUTIONS) onepass_cnt,
       sum(MULTIPASSES_EXECUTIONS) mpass_cnt
FROM V$SQL s, V$SQL_WORKAREA wa
WHERE s.address = wa.address
GROUP BY sql_text
HAVING sum(ONEPASS_EXECUTIONS+MULTIPASSES_EXECUTIONS)>0;
```

Using the hash value and address of a particular cursor, the following query displays the cursor execution plan, including information about the associated work areas.

```
col "O/1/M" format a10
col name format a20
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_executions) "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'
   AND p.hash_value=3738161960;

OPERATION    OPTIONS  NAME     input(MB) LAST_MEM OPTIMAL_ME ONEPASS_ME O/1/M
------------ -------- -------- --------- -------- ---------- ---------- ------
SELECT STATE
HASH         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
```

You can get the address and hash value from the `V$SQL` view by specifying a pattern in the query. For example:

```
SELECT address, hash_value
  FROM V$SQL
WHERE sql_text LIKE '%my_pattern%';
```

Tuning PGA_AGGREGATE_TARGET

To help you tune the initialization parameter `PGA_AGGREGATE_TARGET`, Oracle provides two PGA advice performance views:

- V$PGA_TARGET_ADVICE
- V$PGA_TARGET_ADVICE_HISTOGRAM

By examining these two views, you no longer need to use an empirical approach to tune the value of PGA_AGGREGATE_TARGET. Instead, you can use the content of these views to determine how key PGA statistics will be impacted if you change the value of PGA_AGGREGATE_TARGET.

In both views, values of PGA_AGGREGATE_TARGET used for the prediction are derived from fractions and multiples of the current value of that parameter, to assess possible higher and lower values. Values used for the prediction range from 10 MB to a maximum of 256 GB.

Oracle generates PGA advice performance views by recording the workload history and then simulating this history for different values of PGA_AGGREGATE_TARGET. The simulation process happens in the background and continuously updates the workload history to produce the simulation result. You can view the result at any time by querying V$PGA_TARGET_ADVICE or V$PGA_TARGET_ADVICE_HISTOGRAM.

To enable automatic generation of PGA advice performance views, make sure the following parameters are set:

- PGA_AGGREGATE_TARGET, to enable automatic PGA memory management. Set the initial value as described in "Setting PGA_AGGREGATE_TARGET Initially" on page 7-44.
- STATISTICS_LEVEL. Set this to TYPICAL (the default) or ALL; setting this parameter to BASIC turns off generation of PGA performance advice views.

The content of these PGA advice performance views is reset at instance start-up or when PGA_AGGREGATE_TARGET is altered.

> **Note:** Simulation cannot include all factors of real execution, so derived statistics might not exactly match up with real performance statistics. You should always monitor the system after changing PGA_AGGREGATE_TARGET, to verify that the new performance is what you expect.

V$PGA_TARGET_ADVICE This view predicts how the statistics cache hit percentage and over allocation count in V$PGASTAT will be impacted if you change the value of the initialization parameter PGA_AGGREGATE_TARGET. Example 7–8 shows a typical query of this view:

Example 7–8 Querying V$PGA_TARGET_ADVICE

```
SELECT round(PGA_TARGET_FOR_ESTIMATE/1024/1024) target_mb,
       ESTD_PGA_CACHE_HIT_PERCENTAGE cache_hit_perc,
       ESTD_OVERALLOC_COUNT
  FROM V$PGA_TARGET_ADVICE;
```

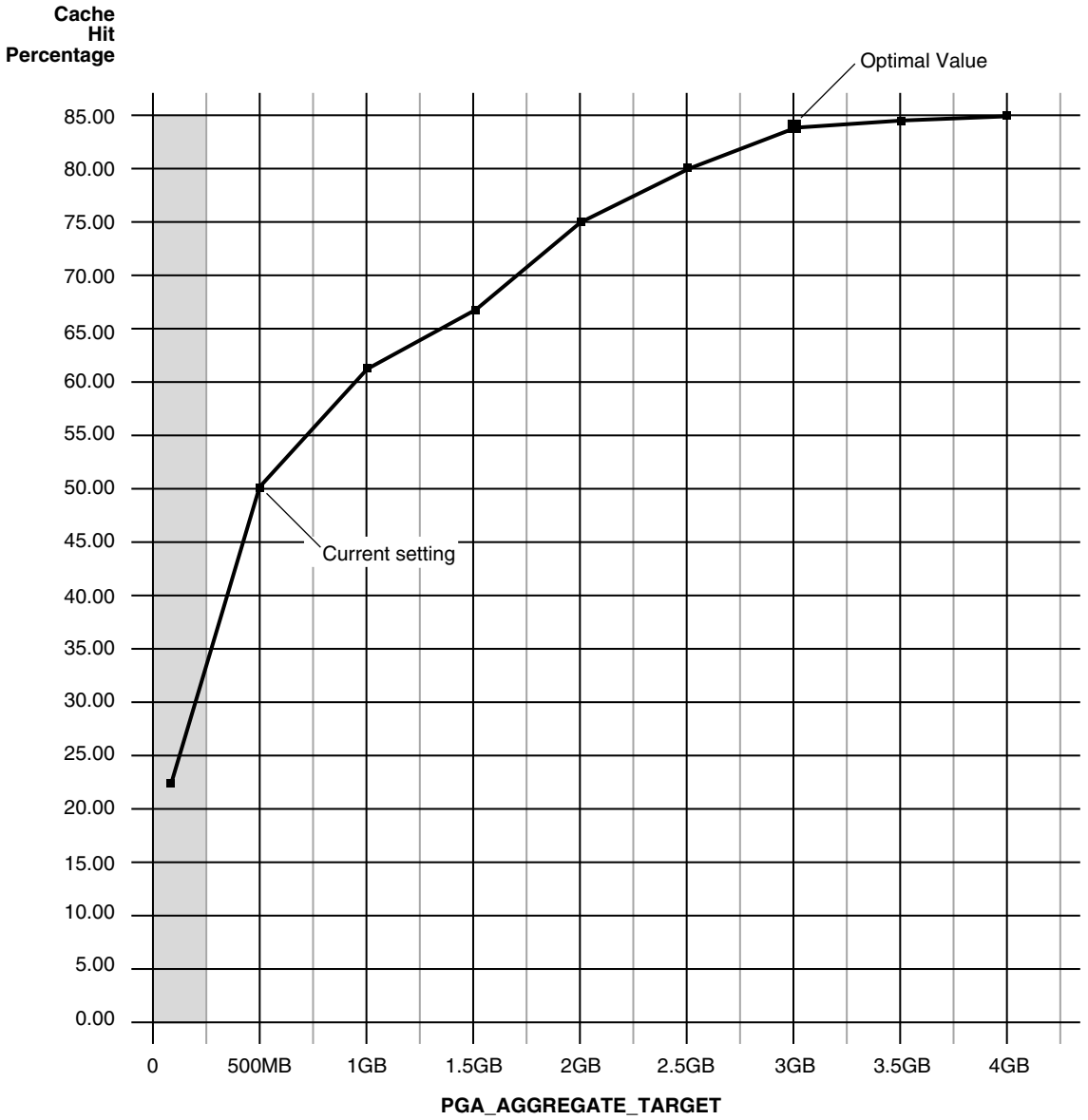
The output of this query might look like the following:

```
 TARGET_MB  CACHE_HIT_PERC  ESTD_OVERALLOC_COUNT
----------  --------------  --------------------
        63              23                   367
       125              24                    30
       250              30                     3
       375              39                     0
       500              58                     0
```

```
   600              59               0
   700              59               0
   800              60               0
   900              60               0
  1000              61               0
  1500              67               0
  2000              76               0
  3000              83               0
  4000              85               0
```

The result of the this query can be plotted as shown in Figure 7–3:

Figure 7–3 Graphical Representation of V$PGA_TARGET_ADVICE



The curve shows how the PGA `cache hit percentage` improves as the value of `PGA_AGGREGATE_TARGET` increases. The shaded zone in the graph is the `over allocation` zone, where the value of the column `ESTD_OVERALLOCATION_COUNT`

is nonzero. It indicates that PGA_AGGREGATE_TARGET is too small to even meet the minimum PGA memory needs. If PGA_AGGREGATE_TARGET is set within the over allocation zone, the memory manager will over-allocate memory and actual PGA memory consumed will be more than the limit you set. It is therefore meaningless to set a value of PGA_AGGREGATE_TARGET in that zone. In this particular example PGA_AGGREGATE_TARGET should be set to at least 375 MB.

Note: Even though the theoretical maximum for the PGA cache hit percentage is 100%, there is a practical limit on the maximum size of a work area, which may prevent this theoretical maximum from being reached, even if you further increase PGA_AGGREGATE_TARGET. This should happen only in large DSS systems where the optimal memory requirement is large and might cause the value of the cache hit percentage to taper off at a lower percentage, like 90%.

Beyond the over allocation zone, the value of the PGA cache hit percentage increases rapidly. This is due to an increase in the number of work areas which run optimally or one-pass and a decrease in the number of multi-pass executions. At some point, somewhere around 500 MB in this example, there is an inflection in the curve that corresponds to the point where most (probably all) work areas can run optimally or at least one-pass. After this inflection, the cache hit percentage keeps increasing, though at a lower pace, up to the point where it starts to taper off and shows only slight improvement with increase in PGA_AGGREGATE_TARGET. In Figure 7–3, this happens when PGA_AGGREGATE_TARGET reaches 3 GB. At that point, the cache hit percentage is 83% and only improves marginally (by 2%) with one extra gigabyte of PGA memory. In this particular example, 3 GB is probably the optimal value for PGA_AGGREGATE_TARGET.

Ideally, PGA_AGGREGATE_TARGET should be set at the optimal value, or at least to the maximum value possible in the region beyond the over allocation zone. As a rule of thumb, the PGA cache hit percentage should be higher than 60%, because at 60% the system is almost processing double the number of bytes it actually needs to process in an ideal situation. Using this particular example, it makes sense to set PGA_AGGREGATE_TARGET to at least 500 MB and as close as possible to 3 GB. But the right setting for the parameter PGA_AGGREGATE_TARGET really depends on how much memory can be dedicated to the PGA component. Generally, adding PGA memory requires reducing memory for some of the SGA components, like the shared pool or the buffer cache. This is because the overall memory dedicated to the Oracle instance is often bound by the amount of physical memory available on the system. As a result, any decisions to increase PGA memory must be taken in the larger context of the available memory in the system and the performance of the various SGA components (which you monitor with shared pool advisory and buffer cache advisory statistics). If memory cannot be taken away from the SGA, you might consider adding more physical memory to the system.

See Also:

- "Shared Pool Advisory Statistics" on page 7-26
- "Sizing the Buffer Cache" on page 7-7

How to Tune PGA_AGGREGATE_TARGET You can use the following steps as a tuning guideline in tuning PGA_AGGREGATE_TARGET:

1. Set PGA_AGGREGATE_TARGET so there is no memory over-allocation; avoid setting it in the over-allocation zone. In Example 7–8, PGA_AGGREGATE_TARGET should be set to at least 375 MB.
2. After eliminating over-allocations, aim at maximizing the PGA cache hit percentage, based on your response-time requirement and memory constraints. In Example 7–8, assume you have a limit *X* on memory you can allocate to PGA.
   - If this limit *X* is beyond the optimal value, then you would set PGA_AGGREGATE_TARGET to the optimal value. After this point, the incremental benefit with higher memory allocation to PGA_AGGREGATE_TARGET is very small. In Example 7–8, if you have 10 GB to dedicate to PGA, set PGA_AGGREGATE_TARGET to 3 GB, the optimal value. The remaining 7 GB is dedicated to the SGA.
   - If the limit *X* is less than the optimal value, then you would set PGA_AGGREGATE_TARGET to *X*. In Example 7–8, if you have only 2 GB to dedicate to PGA, set PGA_AGGREGATE_TARGET to 2 GB and accept a cache hit percentage of 75%.

Finally, like most statistics collected by Oracle that are cumulative since instance start-up, you can take a snapshot of the view at the beginning and at the end of a time interval. You can then derive the predicted statistics for that time interval as follows:

```
        estd_overalloc_count = (difference in estd_overalloc_count between the two snapshots)

                                    (difference in bytes_processed between the two snapshots)
estd_pga_cache_hit_percentage = ------------------------------------------------------------------
                                (difference in bytes_processed + extra_bytes_rw  between the two snapshots )
```

V$PGA_TARGET_ADVICE_HISTOGRAM This view predicts how the statistics displayed by the performance view V$SQL_WORKAREA_HISTOGRAM will be impacted if you change the value of the initialization parameter PGA_AGGREGATE_TARGET. You can use the dynamic view V$PGA_TARGET_ADVICE_HISTOGRAM to view detailed information on the predicted number of optimal, one-pass and multi-pass work area executions for the set of PGA_AGGREGATE_TARGET values you use for the prediction.

The V$PGA_TARGET_ADVICE_HISTOGRAM view is identical to the V$SQL_WORKAREA_HISTOGRAM view, with two additional columns to represent the PGA_AGGREGATE_TARGET values used for the prediction. Therefore, any query executed against the V$SQL_WORKAREA_HISTOGRAM view can be used on this view, with an additional predicate to select the desired value of PGA_AGGREGATE_TARGET.

Example 7–9 Querying V$PGA_TARGET_ADVICE_HISTOGRAM

The following query displays the predicted content of V$SQL_WORKAREA_HISTOGRAM for a value of the initialization parameter PGA_AGGREGATE_TARGET set to twice its current value.

```
SELECT LOW_OPTIMAL_SIZE/1024 low_kb, (HIGH_OPTIMAL_SIZE+1)/1024 high_kb,
       estd_optimal_executions estd_opt_cnt,
       estd_onepass_executions estd_onepass_cnt,
       estd_multipasses_executions estd_mpass_cnt
  FROM v$pga_target_advice_histogram
 WHERE pga_target_factor = 2
   AND estd_total_executions != 0
 ORDER BY 1;
```

The output of this query might look like the following.

```
LOW_KB   HIGH_KB   ESTD_OPTIMAL_CNT   ESTD_ONEPASS_CNT   ESTD_MPASS_CNT
------   -------   ----------------   ----------------   --------------
     8        16             156107                  0                0
    16        32                148                  0                0
    32        64                 89                  0                0
    64       128                 13                  0                0
   128       256                 58                  0                0
   256       512                 10                  0                0
   512      1024                653                  0                0
  1024      2048                530                  0                0
  2048      4096                509                  0                0
  4096      8192                227                  0                0
  8192     16384                176                  0                0
 16384     32768                133                 16                0
 32768     65536                 66                103                0
 65536    131072                 15                 47                0
131072    262144                  0                 48                0
262144    524288                  0                 23                0
```

The output shows that increasing PGA_AGGREGATE_TARGET by a factor of 2 will allow all work areas under 16 MB to execute in optimal mode.

See Also: *Oracle 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
```

Configuring OLAP_PAGE_POOL_SIZE

The OLAP_PAGE_POOL_SIZE initialization parameter specifies (in bytes) the maximum size of the paging cache to be allocated to an OLAP session.

For performance reasons, it is usually preferable to configure a small OLAP paging cache and set a larger default buffer pool with DB_CACHE_SIZE. An OLAP paging cache of 4 MB is fairly typical, with 2 MB used for systems with limited memory resources.

See Also: *Oracle OLAP User's Guide*

Using the Client Query Result Cache

To improve the response time of repetitive queries, Oracle Call Interface (OCI) applications can utilize client memory to take advantage of the OCI result cache. A result cache stores the results of queries shared across all sessions. When these queries are executed repeatedly, the results are retrieved directly from the cache memory. This results in faster response time for the queries. The query results stored in the cache become invalid when data in the database objects being accessed by the query is modified. The client-side result cache is a separate feature from the server-side result cache. Unlike the server result cache, the OCI result cache does not cache results in the server SGA.

This section contains the following topics:

- Using the Result Cache Mode
- Managing the Query Result Cache
- Accessing Information in the Query Result Cache

> **See Also:** *Oracle Call Interface Programmer's Guide* for details about the client result cache and the different settings of client and server cache

Using the Result Cache Mode

You can control the result cache mechanism using the `RESULT_CACHE_MODE` parameter in the server initialization (`init.ora`) file. This parameter can be set at the system, session, or table level to the following values:

- `MANUAL`

  In `MANUAL` mode, the results of a query can only be stored in the result cache by using the `/*+ result_cache */` hint.

- `FORCE`

  In `FORCE` mode, all results are stored in the result cache.

If the `RESULT_CACHE_MODE` parameter is set to `MANUAL`, and you want to store the results of a query in the result cache, then you must explicitly use the `/*+ result_cache */` hint in your query, as shown in the following example:

```
SELECT /*+ result_cache */ deptno, avg(sal)
  FROM emp
 GROUP BY deptno;
```

When you execute the query, OCI will look up the result cache memory to verify if the result for the query already exists in the OCI client cache. If it exists, then the result is retrieved directly out of the cache. Otherwise, the query is executed on the database, and the result is returned as output and stored in the client result cache memory.

If the `RESULT_CACHE_MODE` parameter is set to `FORCE`, and you do not wish to include the result of the query in the result cache, then you must use the `/*+ no_result_cache */` hint in your query, as shown in the following example:

```
SELECT /*+ no_result_cache */ deptno, avg(sal)
  FROM emp
 GROUP BY deptno;
```

> **See Also:** *Oracle Database SQL Reference*

Managing the Query Result Cache

To manage the client result cache, you can alter the following parameter settings in the server initialization file:

- `CLIENT_RESULT_CACHE_SIZE`

  You can change the memory allocated to the result cache by setting this parameter. If you set the value to 0 (the default value), the result cache will be disabled.

- `CLIENT_RESULT_CACHE_LAG`

  You can change the amount of lag time for the client result cache by setting this parameter. If the OCI application does not access the database frequently, setting this parameter to a low value will result in more round trips from the OCI client library to the database in order to keep the client result cache synchronized with the database.

See Also:

- *Oracle Database Reference* for details about these server initialization parameters
- *Oracle Call Interface Programmer's Guide* for recommendations on setting the cache size value

Accessing Information in the Query Result Cache

The `CLIENT_RESULT_CACHE_STATS$` view lists the various cache settings and memory usage statistics for the result cache.

See Also: *Oracle Database Reference* for details about the `CLIENT_RESULT_CACHE_STATS$` view

8

I/O Configuration and Design

The 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

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 if an application is well written, its 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 or near 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).

Consider the following database requirements when designing an I/O system:

- Storage, such as minimum disk capacity
- Availability, such as continuous (24 x 7) or business hours only
- 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 to be configured should be determined by I/O throughput and redundancy requirements. Then, the size of disks can be determined by the storage requirements.

I/O Calibration

The I/O calibration feature of Oracle Database enables you to assess the performance of the storage subsystem, and determine whether I/O performance problems are caused by the database or the storage subsystem. Unlike other external I/O calibration tools that issue I/Os sequentially, the I/O calibration feature of Oracle Database issues I/Os randomly using Oracle datafiles to access the storage media, producing results that more closely match the actual performance of the database.

The section describes how to use the I/O calibration feature of Oracle Database and contains the following topics:

- Prerequisites for I/O Calibration
- Running I/O Calibration

Prerequisites for I/O Calibration

Before running I/O calibration, ensure that the following requirements are met:

- The user must be granted the SYSDBA privilege
- `timed_statistics` must be set to `TRUE`
- Asynchronous I/O must be enabled

  When using file systems, asynchronous I/O can be enabled by setting `filesystemio_options` to `SETALL`
- Ensure that asynchronous I/O is enabled for datafiles by running the following query:

```
col name format a50
select name,asynch_io from v$datafile f,v$iostat_file i
where f.file#=i.file_no
and filetype_name='Data File';
```

Additionally, only one calibration can be performed on a database instance at a time.

Running I/O Calibration

The I/O calibration feature of Oracle Database is accessed using the `DBMS_RESOURCE_MANAGER.CALIBRATE_IO` procedure. This procedure issues an I/O intensive read-only workload (made up of one megabytes of random of I/Os) to the database files to determine the maximum IOPS (I/O requests per second) and MBPS (megabytes of I/O per second) that can be sustained by the storage subsystem. Due to the overhead from running the I/O workload, I/O calibration should only be performed when the database is idle, or during off-peak hours, to minimize the impact of the I/O workload on the normal database workload.

To run I/O calibration and assess the I/O capability of the storage subsystem used by Oracle Database, use the `DBMS_RESOURCE_MANAGER.CALIBRATE_IO` procedure:

```
SET SERVEROUTPUT ON
DECLARE
  lat  INTEGER;
  iops INTEGER;
  mbps INTEGER;
BEGIN
-- DBMS_RESOURCE_MANAGER.CALIBRATE_IO (<DISKS>, <MAX_LATENCY>, iops, mbps, lat);
   DBMS_RESOURCE_MANAGER.CALIBRATE_IO (2, 10, iops, mbps, lat);

  DBMS_OUTPUT.PUT_LINE ('max_iops = ' || iops);
  DBMS_OUTPUT.PUT_LINE ('latency  = ' || lat);
  dbms_output.put_line('max_mbps = ' || mbps);
end;
/
```

When running the `DBMS_RESOURCE_MANAGER.CALIBRATE_IO` procedure, consider the following:

- Do not run the procedure multiple times across separate databases that use the same storage subsystem
- Quiesce the database to minimize I/O on the instance
- For Oracle Real Application Clusters (RAC) configurations, ensure that all instances are opened to calibrate the storage subsystem across nodes
- The execution time of the procedure is dependent on the number of disks in the storage subsystem and increases with the number of nodes in the database
- In some cases, asynchronous I/O is permitted for datafiles, but the I/O subsystem for submitting asynchronous I/O may be maximized, and I/O calibration cannot continue. In such cases, refer to the port-specific documentation for information about checking the maximum limit for asynchronous I/O on the system

At any time during the I/O calibration process, you can query the calibration status in the `V$IO_CALIBRATION_STATUS` view. After I/O calibration is successfully completed, you can view the results in the `DBA_RSRC_IO_CALIBRATE` table.

See Also:

- *Oracle Database PL/SQL Packages and Types Reference* for more information about running the `DBMS_RESOURCE_MANAGER.CALIBRATE_IO` procedure
- *Oracle Database Reference* for information about the `V$IO_CALIBRATION_STATUS` view and `DBA_RSRC_IO_CALIBRATE` table

I/O Configuration

This section describes the basic information to be gathered and decisions to be made when defining a 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.

This section contains the following topics:

Lay Out the Files Using Operating System or Hardware Striping

If your operating system has 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 **stripe depth** and **stripe width**.

- Stripe depth is the size of the stripe, sometimes called stripe unit.
- Stripe width is the product of the stripe depth and the number of drives in the striped set.

Choose these values wisely so that the system is capable of sustaining the required throughput. For an Oracle database, reasonable stripe depths range from 256 KB to 1 MB. Different types of applications benefit from different stripe depths. The optimal stripe depth and stripe width depend 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

Table 8–1 lists the Oracle and operating system parameters that you can use to set I/O size:

Table 8–1 Oracle and Operating System Operational Parameters

Parameter	Description
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). If this value is not set explicitly (or is set to 0), the optimizer will use a default value of 8 for this parameter when calculating the maximum I/O size for full table scans.
SORT_AREA_SIZE	Determines I/O sizes and concurrency for sort operations.
HASH_AREA_SIZE	Determines the I/O size for hash operations.

In addition to I/O size, the degree of concurrency also helps in determining the ideal stripe depth. Consider the following when choosing stripe width and stripe depth:

- On low-concurrency (sequential) 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 depth is 32k. If a single 1MB I/O request (for example, for a full table scan) is issued by an Oracle server process, then each disk in the stripe must perform eight 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 depth. 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 (random) 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 with a high degree of concurrent small I/O requests, such as in a traditional OLTP environment, it is beneficial to keep the stripe depth large. Using stripe depths larger than the I/O size is called coarse grain striping. In high-concurrency systems, the stripe depth can be

```
n * DB_BLOCK_SIZE
```

where *n* is greater than 1.

Coarse grain striping allows a disk in the array to service several I/O requests. In this way, a large number of concurrent I/O requests can be serviced by a set of striped disks with minimal I/O setup costs. Coarse grain striping strives to maximize overall I/O throughput. multiblock reads, as in full table scans, will benefit when stripe depths are large and can be serviced from one drive. Parallel query in a DSS environment is also a candidate for coarse grain striping. This is because there are many individual processes, each issuing separate I/Os. If coarse grain striping is used in systems that do not have high concurrent requests, then hot spots could result.

In a system with a few large I/O requests, such as in a traditional DSS environment or a low-concurrency OLTP system, then it is beneficial to keep the stripe depth small. This is called fine grain striping. In such systems, the stripe depth is

```
n * DB_BLOCK_SIZE
```

where *n* is smaller than the multiblock read parameters, such as `DB_FILE_MULTIBLOCK_READ_COUNT`.

Fine grain striping allows a single I/O request to be serviced by multiple disks. Fine grain striping strives to maximize performance for individual I/O requests or response time.

Alignment of Physical Stripe Boundaries with Block Size Boundaries

On some Oracle ports, an Oracle block boundary may not align with the stripe. If your stripe depth is the same size as the Oracle block, then a single I/O issued by Oracle might result in two physical I/O operations.

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 depth should be at least twice the Oracle block size. Table 8–2 shows recommended minimum stripe depth for random access and for sequential reads.

Table 8–2 Minimum Stripe Depth

Disk Access	Minimum Stripe Depth
Random reads and writes	The minimum stripe depth is twice the Oracle block size.
Sequential reads	The minimum stripe depth is twice the value of `DB_FILE_MULTIBLOCK_READ_COUNT`, multiplied by the Oracle block size.

See Also: The specific documentation for your platform

Manageability of the Proposed System

With an LVM, the simplest configuration to manage is one with 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 viable only 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, performing 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 16 GB. This is total disk space of 1 terabyte (TB) 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 might develop, 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, more manageable stripe width. Then, when new disks are added, the system can grow by a stripe width.

In the preceding example, eight disks might be a more manageable stripe width. This is only feasible if eight disks are capable of sustaining the required number of I/Os each 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 will 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 workload to analyze, you can 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. Evaluate database disk-storage requirements by checking the size of the files and the disks.
2. Identify the expected I/O throughput for each 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 approach to manual I/O distribution 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 be made only when the I/O rate for that file affects database performance.

When to Separate Files

Regardless of whether you use operating system striping or manual I/O distribution, if the I/O system or I/O layout is not able to support the I/O rate required, then you need to separate files with high I/O rates from the remaining files. You can identify such files 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 a 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.

The following sections describe how to segregate the following file types:

- Tables, Indexes, and TEMP Tablespaces
- Redo Log Files
- Archived Redo Logs

See Also: "Identifying High-Load SQL" on page 16-2

Tables, Indexes, and TEMP Tablespaces

If the files with high I/O 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 files with high-I/O are datafiles that belong to the `TEMP` tablespace, then investigate whether to tune the SQL statements performing disk sorts to avoid this activity, or to 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 High-Load SQL" on page 16-2

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:

- Placing 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.
- Placing each redo log group on a separate disk that does not store any other files.
- Striping the redo log files across several disks, using an operating system striping tool. (Manual striping is not possible in this situation.)
- Avoiding the use of RAID 5 for redo logs.

Redo log files are written sequentially by the Log Writer (LGWR) process. 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.

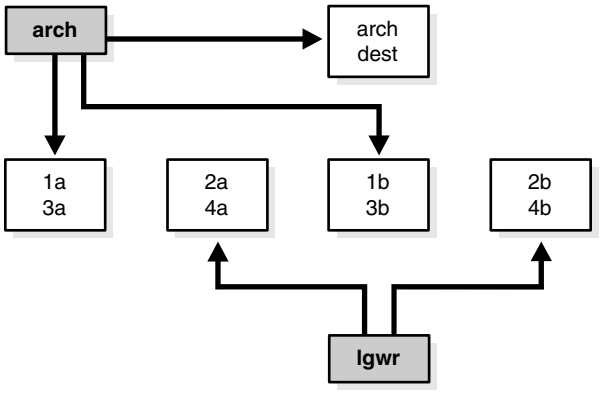
Archived Redo Logs

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, suppose a system has four redo log groups, each group with two members. To create separate-disk access, the eight log files should be labeled 1a, 1b, 2a, 2b, 3a, 3b, 4a, and 4b. This requires at least four disks, plus one disk for archived files.

Figure 8–1 illustrates how redo members should be distributed across disks to minimize contention.

Figure 8–1 Distributing Redo Members Across Disks



In this example, LGWR switches out of log group 1 (member 1a and 1b) and writes to log group 2 (2a and 2b). Concurrently, the archiver process reads from group 1 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

This section contains three high-level examples of configuring I/O systems. These examples include sample calculations that define the disk topology, stripe depths, and so on:

Stripe Everything Across Every Disk

The simplest approach to I/O configuration is to build one giant volume, striped across all available disks. To account for recoverability, the volume is mirrored (RAID 1). The striping unit for each 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 could suffer when redo logs are being archived. Moving archive logs to separate disks provides the following benefits:

- The archive can be performed at very high rate (using 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 that are separate from other disks and from archived redo log files has the following benefits:

- Writing redo logs is performed at the highest possible rate. 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 not 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, temp files, online logs, and control files. Administrators only specify the file system directory to be used for a particular type of file. You can specify one default location for datafiles and up to five multiplexed locations for the control and online redo log files.

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.

Several points should be considered when 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 an LVM 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 an LVM that supports dynamically extensible logical volumes. Otherwise, the logical volumes should be configured as large as possible.
- Oracle-managed files work best if the file system provides large extensible files.

> **See Also:** *Oracle Database Administrator's Guide* for detailed information on using Oracle-managed files

Choosing Data Block Size

A block size of 8K is optimal for most systems. However, OLTP systems occasionally use smaller block sizes and DSS systems occasionally use larger block sizes. This section discusses considerations when choosing database block size for optimal performance and contains the following topics:

- Reads
- Writes
- Block Size Advantages and Disadvantages

> **Note:** The use of multiple block sizes in a single database instance is not encouraged because of manageability issues.

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, such as rows containing large object (LOB) data, then choose a larger block size.

Writes

For high-concurrency OLTP systems, consider appropriate values for `INITRANS`, `MAXTRANS`, and `FREELISTS` when using a larger block size. These parameters affect the degree of update concurrency allowed within a block. However, you do not need to specify the value for `FREELISTS` when using automatic segment-space management.

If you are uncertain about which block size to choose, then try a database block size of 8 KB for most systems that process a large number of transactions. This represents a good compromise and is usually effective. Only systems processing LOB data need more than 8 KB.

> **See Also:** The Oracle documentation specific to your operating system for information on the minimum and maximum block size on your platform

Block Size Advantages and Disadvantages

Table 8–3 lists the advantages and disadvantages of different block sizes.

Table 8–3 Block Size Advantages and Disadvantages

Block Size	Advantages	Disadvantages
Smaller	Good for small rows with lots of random access.	Has relatively large space overhead due to metadata (that is, block header).
	Reduces block contention.	Not recommended for large rows. There might only be a few rows stored for each block, or worse, row chaining if a single row does not fit into a block,
Larger	Has lower overhead, so there is more room to store data.	Wastes space in the buffer cache, if you are doing random access to small rows and have a large block size. For example, with an 8 KB block size and 50 byte row size, you waste 7,950 bytes in the buffer cache when doing random access.
	Permits reading 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).	Not good for index blocks used in an OLTP environment, because they increase block contention on the index leaf blocks.

9

Understanding Operating System Resources

This chapter explains how to tune the operating system for optimal performance of the Oracle database 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 contains tuning information specific to your platform. Your operating system vendor' s documentation should also be consulted.
- "Operating System Statistics" on page 5-4 for a discussion of the importance of operating system statistics

Understanding Operating System Performance Issues

Operating system performance issues commonly involve process management, memory management, and scheduling. If you have tuned the Oracle instance and still need to improve performance, verify your work or try to reduce system time. Ensure 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 result in 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.

This section covers the following topics related to operating system performance issues:

- Using Operating System Caches
- Memory Usage
- Using Operating System Resource Managers

Using Operating System Caches

Operating systems and device controllers provide data caches that do not directly conflict with Oracle cache management. Nonetheless, these structures can consume resources while offering little or no benefit to performance. This is most noticeable on a UNIX system that has the database files in the UNIX file store; by default, all database I/O goes through the file system cache. On some UNIX systems, direct I/O is available to the filestore. This arrangement allows the database files to be accessed within the UNIX file system, bypassing the file system cache. It saves CPU resources and allows the file system cache to be dedicated to non-database activity, such as program texts and spool files.

This problem does not occur on Windows. All file requests by the database bypass the caches in the file system.

Although the operating system cache is often redundant because the Oracle buffer cache buffers blocks, there are a number of cases where Oracle does not use the Oracle buffer cache. In these cases, using direct I/O which bypasses the Unix or operating system cache, or using raw devices which do not use the operating system cache, may yield worse performance than using operating system buffering. Some examples include:

- Reads or writes to the `TEMPORARY` tablespace
- Data stored in `NOCACHE` LOBs
- Parallel Query slaves reading data

You may want a mix with some files cached at the operating system level and others not.

Asynchronous I/O

With synchronous I/O, when an I/O request is submitted to the operating system, the writing process blocks until the write is confirmed as complete. It can then continue processing. With asynchronous I/O, processing continues while the I/O request is submitted and processed. Use asynchronous I/O when possible to avoid bottlenecks.

Some platforms support asynchronous I/O by default, others need special configuration, and some only support asynchronous I/O for certain underlying file system types.

FILESYSTEMIO_OPTIONS Initialization Parameter

You can use the `FILESYSTEMIO_OPTIONS` initialization parameter to enable or disable asynchronous I/O or direct I/O on file system files. This parameter is platform-specific and has a default value that is best for a particular platform. It can be dynamically changed to update the default setting.

`FILESYTEMIO_OPTIONS` can be set to one of the following values:

- `ASYNCH`: enable asynchronous I/O on file system files, which has no timing requirement for transmission.
- `DIRECTIO`: enable direct I/O on file system files, which bypasses the buffer cache.
- `SETALL`: enable both asynchronous and direct I/O on file system files.
- `NONE`: disable both asynchronous and direct I/O on file system files.

See Also: Your platform-specific documentation for more details

Memory Usage

Memory usage is affected by both buffer cache limits and initialization parameters.

Buffer Cache Limits

The UNIX buffer cache consumes operating system memory resources. Although in some versions of UNIX, the UNIX buffer cache may be allocated a set amount of memory, it is common today for more sophisticated memory management mechanisms to be used. Typically, these will allow free memory pages to be used to cache I/O. In such systems, it is common for operating system reporting tools to show that there is no free memory, which is not generally a problem. If processes require more memory, the memory caching I/O data is usually released to allow the process memory to be allocated.

Parameters Affecting Memory Usage

The memory required by any one Oracle session depends on many factors. Typically the major contributing factors are:

- Number of open cursors
- Memory used by PL/SQL, such as PL/SQL tables
- `SORT_AREA_SIZE` initialization parameter

In Oracle, the `PGA_AGGREGATE_TARGET` initialization parameter gives greater control over a session's memory usage.

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 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, as long as the allocation of partitioned memory (RAM) resources is fixed, not dynamic.

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.

Note: Oracle is not supported for use with any UNIX operating system resource manager's memory management and allocation facility. Oracle Database Resource Manager, which provides resource allocation capabilities within an Oracle instance, cannot be used with any operating system resource manager.

Caution: 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.

See Also:

- For a complete list of operating system resource management and resource allocation and 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.
- *Oracle 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 Windows Systems
- Performance Hints on HP OpenVMS Systems

Familiarize yourself with platform-specific issues so that you know what performance options the operating system provides.

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 most of the time in application mode, also called user mode, rather than system mode.

The ratio of time spent in each mode is only a symptom of the underlying problem, which might involve the following:

- Paging or swapping
- Executing too many operating 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 Windows Systems

On Windows systems, as with UNIX-based systems, establish an appropriate ratio between time in application mode and time in system mode. You can easily monitor many factors with the Windows administrative performance tool: 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 HP OpenVMS Systems

Consider the paging parameters on a mainframe, and remember that Oracle can exploit a very large working set.

Free memory in HP OpenVMS 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.

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 the Automatic Workload Repository, Statspack, or the `UTLBSTAT/UTLESTAT` utility. Operating system utilities—such as `vmstat`, `sar`, and `iostat` on UNIX and the administrative performance monitoring tool on Windows—can be used along with the `V$OSSTAT` or `V$SYSMETRIC_HISTORY` view during the same time interval as Automatic Workload Repository, Statspack, or `UTLBSTAT/UTLESTAT` to provide a complimentary view of the overall statistics.

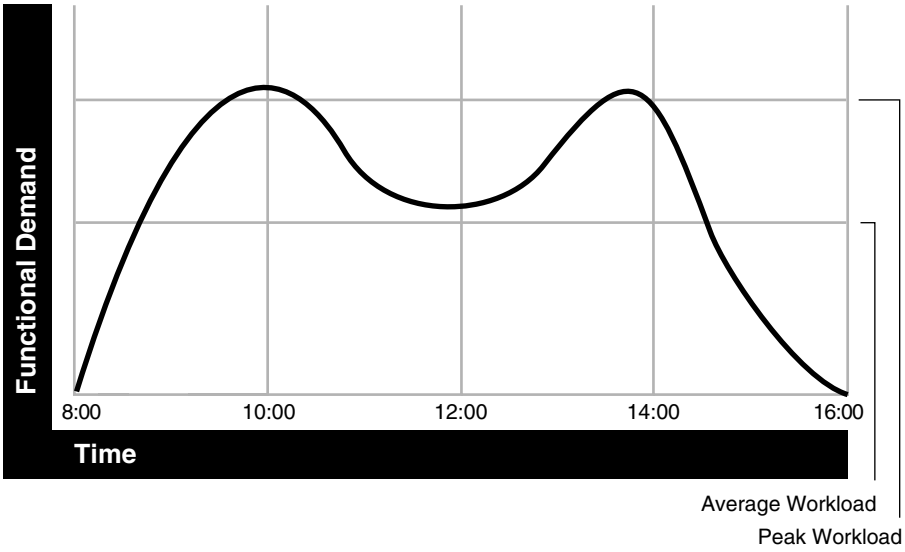
See Also:

- "Overview of the Automatic Workload Repository" on page 5-8
- Chapter 6, "Automatic Performance Diagnostics" for more information on Oracle tools

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 9–1 illustrates workload over time for an application having peak periods at 10:00 AM and 2:00 PM.

Figure 9–1 Average Workload and Peak Workload



This example application has 100 users working 8 hours a 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 an hour, which is an average of 20 transactions a 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 a minute can be understood as merely a minimum requirement. If the peak rate you need to achieve is 120 transactions a 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% of available CPU resource
```

where `tpm` is transactions a minute.

If the system requires 50% of the CPU resource to achieve 20 tpm, then a problem exists: the system cannot achieve 120 transactions a 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 a 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:

- Tuning, or the process of detecting and solving CPU problems from excessive consumption. See "Finding System CPU Utilization" on page 9-7.
- Increasing hardware capacity, including changing the system architecture

See Also: "System Architecture" on page 2-5 for information about improving your system architecture

- Reducing the impact of peak load use patterns by prioritizing CPU resource allocation. Oracle Database Resource Manager does this by allocating and managing CPU resources among database users and applications in the following ways:
  - Limit number of active sessions for each Consumer Group

    This feature is particularly important when a Consumer Group has a lot of parallel queries and you want to limit the total number of parallel queries.
  - CPU saturation

    If the CPUs are running at 100%, Oracle Database Resource Manager can be used to allocate a minimum amount of CPU to sessions in each Consumer Group. This feature can lower the CPU consumption of low-priority sessions.
  - Runaway queries

    Oracle Database Resource Manager can limit the damage from runaway queries by limiting the maximum execution time for a call, or by moving the long-running query to a lower priority Consumer Group.

    See Also: *Oracle Database Administrator's Guide* for more information about Oracle Database Resource Manager

Finding System CPU Utilization

Every process running on your system affects the available CPU resources. Therefore, tuning non-Oracle factors can also improve Oracle performance.

Use the `V$OSSTAT` or `V$SYSMETRIC_HISTORY` view to monitor system utilization statistics from the operating system. Useful statistics contained in `V$OSSTAT` and `V$SYSMETRIC_HISTORY` include:

- Number of CPUs
- CPU utilization
- Load
- Paging
- Physical memory

See Also: *Oracle Database Reference* for more information on `V$OSSTAT` and `V$SYSMETRIC_HISTORY`.

Operating system monitoring tools can be used 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 allow you to 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 Windows, use the administrative performance tool to monitor CPU utilization. This utility provides statistics on processor time, user time, privileged time, interrupt time, and DPC time.

This section contains the following topics related to checking system CPU utilization:

- Checking Memory Management
- Checking I/O Management
- Checking Network Management
- Checking Process Management

Note: This section describes how to check system CPU utilization on most UNIX-based and Windows systems. For other platforms, see your operating system documentation.

Checking Memory Management

Check the following memory management areas:

Paging and Swapping

Use the `V$OSSTAT` view, utilities such as `sar` or `vmstat` on UNIX, or the administrative performance tool on Windows, 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 Windows systems.

Checking I/O Management

Thrashing is an I/O management issue. 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.

See Also: Chapter 8, "I/O Configuration and Design"

Checking Network Management

Check client/server round trips. There is an overhead in processing messages. When an application generates many messages that need to be sent through the network, the latency of sending a message can result in CPU overload. To alleviate this problem, bundle multiple messages together rather than perform lots of round trips. For example, you can use array inserts, array fetches, and so on.

Checking Process Management

Several process management issues discussed in this section should be checked.

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 it is possible that too many processes are being used. On Windows systems, do not overload the 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 Windows, which has only one process for each instance. All threads share the same page table.

Oracle has the several features for context switching, described in this section.

Post-wait Driver An Oracle process needs to be able to post another Oracle process (give it a message) and also needs to be able to wait to be posted.

For example, a foreground process may need to post LGWR to tell it to write out all blocks up to a given point so that it can acknowledge a commit.

Often this post-wait mechanism is implemented through UNIX Semaphores, but these can be resource intensive. Therefore, some platforms supply a post-wait driver, typically a kernel device driver that is a lightweight method of implementing a post-wait interface.

Memory-mapped System Timer Oracle often needs to query the system time for timing information. This can involve an operating system call that incurs a relatively costly context switch. Some platforms implement a memory-mapped timer that uses an address within the processes virtual address space to contain the current time information. Reading the time from this memory-mapped timer is less expensive than the overhead of a context switch for a system call.

List I/O Interfaces to Submit Multiple Asynchronous I/Os in One Call List I/O is an application programming interface that allows several asynchronous I/O requests to be submitted in a single system call, rather than submitting several I/O requests through separate system calls. The main benefit of this feature is to reduce the number of context switches required.

Starting New Operating System Processes

There is a high cost in starting new operating system processes. 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 4 KB, and a page table entry might be 8 bytes. You could end up with (1G / 4 KB) * 8 byte entries. This becomes expensive, because you need to continually make sure that the page table is loaded.

10

Instance Tuning Using Performance Views

After the initial configuration of a database, monitoring and tuning an instance regularly is important to eliminate any potential performance bottlenecks. This chapter discusses the tuning process using Oracle V$ performance views.

This chapter contains the following sections:

- Instance Tuning Steps
- Interpreting Oracle Statistics
- Wait Events Statistics
- Real-Time SQL Monitoring

Instance Tuning Steps

These are the main steps in the Oracle performance method for instance tuning:

1. Define the Problem

   Get candid feedback from users about the scope of the performance problem.

2. Examine the Host System and Examine the Oracle Statistics

   - After obtaining a full set of operating system, database, and application statistics, examine the data for any evidence of performance problems.
   - Consider the list of common performance errors to see whether the data gathered suggests that they are contributing to the problem.
   - Build a conceptual model of what is happening on the system using the performance data gathered.

3. Implement and Measure Change

   Propose changes to be made and the expected result of implementing the changes. Then, implement the changes and measure application performance.

4. Determine whether the performance objective defined in step 1 has been met. If not, then repeat steps 2 and 3 until the performance goals are met.

See Also: "The Oracle Performance Improvement Method" on page 3-1 for a theoretical description of this performance method and a list of common errors

The remainder of this chapter discusses instance tuning using the Oracle dynamic performance views. However, Oracle recommends using the Automatic Workload Repository and Automatic Database Diagnostic Monitor for statistics gathering,

monitoring, and tuning due to the extended feature list. See "Overview of the Automatic Workload Repository" on page 5-8 and "Overview of the Automatic Database Diagnostic Monitor" on page 6-1.

> **Note:** If your site does not have the Automatic Workload Repository and Automatic Database Diagnostic Monitor features, then Statspack can be used to gather Oracle instance statistics.

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 an 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 16, "SQL Tuning Overview" 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
- Identifying I/O Problems
- Identifying Network Issues

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. Check whether some processes alone consume the full power of one CPU. Depending on the process, this could be an indication of a CPU or process bound workload which can be tackled by dividing or parallelizing the process activity.

Non-Oracle Processes If the programs are not Oracle programs, then identify whether they are legitimately requiring that amount of CPU. If so, determine whether their execution be delayed to off-peak hours. Identifying these CPU intensive processes can also help narrowing what specific activity, such as I/O, network, and paging, is consuming resources and how can it be related to the Oracle workload.

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 its 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 several `V$` views:

- `V$SYSSTAT` shows Oracle CPU usage for all sessions. The `CPU used by this session` statistic shows the aggregate CPU used by all sessions. The `parse time cpu` statistic shows the total CPU time used for parsing.
- `V$SESSTAT` shows Oracle CPU usage for each session. Use this view to determine which particular session is using the most CPU.
- `V$RSRC_CONSUMER_GROUP` shows CPU utilization statistics for each consumer group 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 Windows and UNIX, this can be either user time or system time (privileged mode on Windows). Thus, average CPU time utilized by all processes (threads) on the system could be greater than one minute for every 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 Chapter 21, "Using Application Tracing Tools".

If the CPU usage is evenly distributed over many Oracle server processes, examine the V$SYS_TIME_MODEL view to help get a precise understanding of where most time is spent. See Table 10–1, " Wait Events and Potential Causes" on page 10-14.

Identifying 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. If the I/O problems are caused by Oracle Database, then I/O tuning can begin. If Oracle Database 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.

I/O problems can be identified using V$ views in Oracle Database and monitoring tools in the operating system, as described in the following sections:

- Identifying I/O Problems Using V$ Views
- Identifying I/O Problems Using Operating System Monitoring Tools

Identifying I/O Problems Using V$ Views 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, db file parallel write, and log file parallel write. These are all events corresponding to I/Os performed against datafiles and log 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 Automatic Repository report to identify hot datafiles and tablespaces. Also compare the I/O times reported by the operating system with the times reported by Oracle to see if they are consistent.

An I/O problem can also manifest itself with non-I/O related wait events. For example, the difficulty in finding a free buffer in the buffer cache or high wait times for logs to be flushed to disk can also be symptoms of an I/O problem. Before investigating whether the I/O system should be reconfigured, determine if the load on the I/O system can be reduced.

To reduce I/O load caused by Oracle Database, examine the I/O statistics collected for all I/O calls made by the database using the following views:

- V$IOSTAT_CONSUMER_GROUP

  The V$IOSTAT_CONSUMER_GROUP view captures I/O statistics for consumer groups. If Oracle Database Resource Manager is enabled, I/O statistics for all consumer groups that are part of the currently enabled resource plan are captured.

- V$IOSTAT_FILE

  The V$IOSTAT_FILE view captures I/O statistics of database files that are or have been accessed. The SMALL_SYNC_READ_LATENCY column displays the latency for single block synchronous reads (in milliseconds), which translates

directly to the amount of time that clients need to wait before moving onto the next operation. This defines the responsiveness of the storage subsystem based on the current load. If there is a high latency for critical datafiles, you may want to consider relocating these files to improve their service time. To calculate latency statistics, `timed_statistics` must be set to `TRUE`.

- `V$IOSTAT_FUNCTION`

  The `V$IOSTAT_FUNCTION` view captures I/O statistics for database functions (such as the LGWR and DBWR).

  An I/O can be issued by various Oracle processes with different functionalities. The top database functions are classified in the `V$IOSTAT_FUNCTION` view. In cases when there is a conflict of I/O functions, the I/O is placed in the bucket with the lower `FUNCTION_ID`. For example, if XDB issues an I/O from the buffer cache, the I/O would be classified as an XDB I/O because it has a lower `FUNCTION_ID` value. Any unclassified function is placed in the Others bucket. You can display the `FUNCTION_ID` hierarchy by querying the `V$IOSTAT_FUNCTION` view:

```
select FUNCTION_ID, FUNCTION_NAME
from v$iostat_function
order by FUNCTION_ID;

FUNCTION_ID FUNCTION_NAME
----------- ------------------
          0 RMAN
          1 DBWR
          2 LGWR
          3 ARCH
          4 XDB
          5 Streams AQ
          6 Data Pump
          7 Recovery
          8 Buffer Cache Reads
          9 Direct Reads
         10 Direct Writes
         11 Others
```

These `V$IOSTAT` views contains I/O statistics for both single and multi block read and write operations. Single block operations are small I/Os that are less than or equal to 128 kilobytes. Multi block operations are large I/Os that are greater than 128 kilobytes. For each of these operations, the following statistics are collected:

- Identifier
- Total wait time (in milliseconds)
- Number of waits executed (for consumer groups and functions)
- Number of requests for each operation
- Number of single and multi block bytes read
- Number of single and multi block bytes written

You should also look at SQL statements that perform many physical reads by querying the `V$SQLAREA` view, or by reviewing the 'SQL ordered by Reads' section of the Automatic Workload Repository report. Examine these statements to see how they can be tuned to reduce the number of I/Os.

See Also:

- Chapter 8, "I/O Configuration and Design"
- Chapter 16, "SQL Tuning Overview"
- "db file scattered read" on page 10-20 and "db file sequential read" on page 10-21 for the difference between a scattered read and a sequential read, and how this affects I/O
- *Oracle Database Reference* for information about the `V$IOSTAT_CONSUMER_GROUP`, `V$IOSTAT_FUNCTION`, `V$IOSTAT_FILE`, and `V$SQLAREA` views

Identifying I/O Problems Using Operating System Monitoring Tools 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 Database. 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.

Utilities, such as `sar -d` (or `iostat`) on many UNIX systems and the administrative performance monitoring tool on Windows systems, examine I/O statistics for the entire system.

See Also: Your operating system documentation for the tools available on your platform

Identifying Network Issues

Using operating system 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.

To identify network I/O caused by remote access of database files, examine the `V$IOSTAT_NETWORK` view. This view contains network I/O statistics caused by accessing files on a remote database instance, including:

- Database client initiating the network I/O (such as RMAN and PLSQL)
- Number of read and write operations issued
- Number of kilobytes read and written
- Total wait time in milliseconds for read operations
- Total wait in milliseconds for write operations

After the cause of the network issue is identified, 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) for one request.

Examine the Oracle Statistics

Oracle statistics should be examined and cross-referenced with operating system statistics to ensure a consistent diagnosis of the problem. operating-system 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 an Oracle perspective before implementing corrective action. See "Interpreting Oracle Statistics" on page 10-11.

The following sections discuss the common Oracle data sources used while tuning.

Setting the Level of Statistics Collection

Oracle provides the initialization parameter STATISTICS_LEVEL, which controls all major statistics collections or advisories in the database. This parameter sets the statistics collection level for the database.

Depending on the setting of STATISTICS_LEVEL, certain advisories or statistics are collected, as follows:

- BASIC: No advisories or statistics are collected. Monitoring and many automatic features are disabled. Oracle does not recommend this setting because it disables important Oracle features.
- TYPICAL: This is the default value and ensures collection for all major statistics while providing best overall database performance. This setting should be adequate for most environments.
- ALL: All of the advisories or statistics that are collected with the TYPICAL setting are included, plus timed operating system statistics and row source execution statistics.

See Also:

- *Oracle Database Reference* for more information on the STATISTICS_LEVEL initialization parameter
- "Interpreting Statistics" on page 5-7 for considerations when setting the STATISTICS_LEVEL, DB_CACHE_ADVICE, TIMED_STATISTICS, or TIMED_OS_STATISTICS initialization parameters

V$STATISTICS_LEVEL This view lists the status of the statistics or advisories controlled by STATISTICS_LEVEL.

See Also: *Oracle Database Reference* for information about the dynamic performance V$STATISTICS_LEVEL view

Wait Events

Wait events are statistics that are incremented by a server process or 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.

Wait events are grouped into classes. The wait event classes include: Administrative, Application, Cluster, Commit, Concurrency, Configuration, Idle, Network, Other, Scheduler, System I/O, and User I/O.

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.

> **See Also:** *Oracle Database Reference* for more information about Oracle wait events

Wait event statistics include the number of times an event was waited for and the time waited for the event to complete. If the initialization 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. See "Setting the Level of Statistics Collection" on page 10-7 for information about STATISTICS_LEVEL settings.

Dynamic Performance Views Containing Wait Event Statistics

These dynamic performance views can be queried for wait event statistics:

- V$ACTIVE_SESSION_HISTORY

  The V$ACTIVE_SESSION_HISTORY view displays active database session activity, sampled once every second. See "Active Session History" on page 5-3.

- V$SESS_TIME_MODEL and V$SYS_TIME_MODEL

  The V$SESS_TIME_MODEL and V$SYS_TIME_MODEL views contain time model statistics, including DB time which is the total time spent in database calls

- V$SESSION_WAIT

  The V$SESSION_WAIT view displays information about the current or last wait for each session (such as wait Id, class, and time).

- V$SESSION

  The V$SESSION view displays information about each current session and contains the same wait statistics as those found in the V$SESSION_WAIT view. If applicable, this view also contains detailed information about the object that the session is currently waiting for (such as object number, block number, file number, and row number), the blocking session responsible for the current wait (such as the blocking session Id, status, and type), and the amount of time waited.

- V$SESSION_EVENT

  The V$SESSION_EVENT view provides summary of all the events the session has waited for since it started.

- V$SESSION_WAIT_CLASS

  The V$SESSION_WAIT_CLASS view provides the number of waits and the time spent in each class of wait events for each session.

- V$SESSION_WAIT_HISTORY

  The V$SESSION_WAIT_HISTORY view displays information about the last ten wait events for each active session (such as event type and wait time).

- V$SYSTEM_EVENT

  The V$SYSTEM_EVENT view provides a summary of all the event waits on the instance since it started.

- V$EVENT_HISTOGRAM

The V$EVENT_HISTOGRAM view displays a histogram of the number of waits, the maximum wait, and total wait time on an event basis.

- V$FILE_HISTOGRAM

  The V$FILE_HISTOGRAM view displays a histogram of times waited during single block reads for each file.

- V$SYSTEM_WAIT_CLASS

  The V$SYSTEM_WAIT_CLASS view provides the instance wide time totals for the number of waits and the time spent in each class of wait events.

- V$TEMP_HISTOGRAM

  The V$TEMP_HISTOGRAM view displays a histogram of times waited during single block reads for each temporary file.

> **See Also:** *Oracle Database Reference* for information about the dynamic performance views

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$SYSTEM_EVENT, you might notice lots of buffer busy waits. It might be that many processes are inserting into the same block and must wait for each other before they can insert. The solution could be to use automatic segment space management or partitioning for the object in question. See "Wait Events Statistics" on page 10-17 for a description of the differences between the views V$SESSION_WAIT, V$SESSION_EVENT, and V$SYSTEM_EVENT.

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$SESSION real-time while the problem is occurring or V$ACTIVE_SESSION_HISTORY and DBA_HIST_ACTIVE_SESS_HISTORY views after the problem has been experienced to identify the contended-for objects using the object 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$ACTIVE_SESSION_HISTORY This view displays active database session activity, sampled once every second. See "Active Session History" on page 5-3.

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 for each file, including the number of I/Os for each file and the average read time.

V$ROLLSTAT This contains detailed rollback and undo segment statistics for each segment.

V$ENQUEUE_STAT This contains detailed enqueue statistics for each enqueue, 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 for each latch, including the number of times each latch was requested and the number of times the latch was waited for.

> **See Also:** *Oracle Database Reference* for information about dynamic performance views

Segment-Level Statistics

You can gather segment-level statistics to help you spot performance problems associated with individual segments. Collecting and viewing segment-level statistics is a good way to effectively identify hot tables or indexes in an instance.

After viewing wait events and system statistics to identify the performance problem, you can use segment-level statistics to find specific tables or indexes that are causing the problem. Consider, for example, that `V$SYSTEM_EVENT` indicates that buffer busy waits cause a fair amount of wait time. You can select from `V$SEGMENT_STATISTICS` the top segments that cause the buffer busy waits. Then you can focus your effort on eliminating the problem in those segments.

You can query segment-level statistics through the following dynamic performance views:

- `V$SEGSTAT_NAME` This view lists the segment statistics being collected, as well as the properties of each statistic (for instance, if it is a sampled statistic).
- `V$SEGSTAT` This is a highly efficient, real-time monitoring view that shows the statistic value, statistic name, and other basic information.
- `V$SEGMENT_STATISTICS` This is a user-friendly view of statistic values. In addition to all the columns of `V$SEGSTAT`, it has information about such things as the segment owner and table space name. It makes the statistics easy to understand, but it is more costly.

> **See Also:** *Oracle Database Reference* for information about dynamic performance views

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 be 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 operating system upgrade if SQL is tuned, the operating system 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: "Overview of Data Gathering" on page 5-1

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 read total bytes`, `physical writes`, `physical write total bytes`, `parse count (total)`, `parse count (hard)`, and `user calls`. This data is queried from `V$SYSSTAT`. It is best to normalize this data over seconds and over transactions. It is also useful to examine the total I/O load in MB per second by using the sum of physical write total bytes and physical write total bytes. The combined value includes the I/O's used to buffer cache, redo logs, archive logs, by RMAN backup and recovery, as well as any Oracle background process.

In the Automatic Workload Repository report, look at the Load Profile section. The data has been normalized over transactions and over seconds.

Changing Load

The load profile statistics over seconds show the changes in throughput (that is, whether the instance is performing more work each second). The statistics over transactions identify changes in the application characteristics by comparing these to the corresponding statistics from the baseline report.

High Rates of Activity

Examine the statistics normalized over seconds 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.

The following are some generalized examples (acceptable values vary at each site):

- A hard parse rate of more than 100 a 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 the sum of the wait times for library cache and shared pool latch events (latch: library cache, latch: library cache pin, latch: library cache lock and latch: shared pool) is significant compared to statistic `DB time` found in `V$SYSSTAT`. If so, examine the `SQL ordered by Parse Calls` section of the Automatic Workload Repository report.
- A high soft parse rate could be in the rate of 300 a second or more. Unnecessary soft parses also limit application scalability. Optimally, a SQL statement should be soft parsed once in each 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. These wait events are grouped in wait classes. The Idle wait class groups all events that a process waits for when it does not have work to do and is waiting for more work to perform. Non-idle 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 10-15 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.

See Also:

- "Setting the Level of Statistics Collection" on page 10-7 for information about `STATISTICS_LEVEL` settings
- *Oracle Database Reference* for information on the `STATISTICS_LEVEL` initialization parameter

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`, and `SQL*Net more data to client`. 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" on page 10-29 for the list of idle wait events
- Description of the V$EVENT_NAME view in *Oracle Database Reference*
- Detailed wait event information in *Oracle Database Reference*

Alternatively, look at the Top 5 Timed Events section at the beginning of the Automatic Workload Repository report. This section automatically orders the wait events (omitting idle events), and calculates the relative percentage:

```
Top 5 Timed Events
~~~~~~~~~~~~~~~~~~                                                 % Total
Event                                            Waits    Time (s) Call Time
------------------------------------------- ------------ ----------- ---------
CPU time                                                         559     88.80
log file parallel write                            2,181          28      4.42
SQL*Net more data from client                    516,611          27      4.24
db file parallel write                            13,383          13      2.04
db file sequential read                              563           2       .27
```

In some situations, there might be a few events with similar percentages. This can provide extra evidence if all the events are 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. The following example of this data is taken from the Wait Event section of the Automatic Workload Repository report:

```
                                                          Avg
                                                         wait     Waits
Event                            Waits  Timeouts   Time (s)   (ms)    /txn
-------------------------- ---------- --------- ---------- ------ --------
log file parallel write         2,181         0         28     13     41.2
SQL*Net more data from clie   516,611         0         27      0 9,747.4
db file parallel write         13,383         0         13      1    252.5
```

3. The top wait events identify the next places to investigate. A table of common wait events is listed in Table 10–1. 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, operating system statistics, and so on. Perform cross-checks with other data to confirm or refute the developing theory.

Table of Wait Events and Potential Causes

Table 10–1 links wait events to possible causes and gives an overview of the Oracle data that could be most useful to review next.

Table 10–1 Wait Events and Potential Causes

Wait Event	General Area	Possible Causes	Look for / Examine
`buffer busy waits`	Buffer cache, DBWR	Depends on buffer type. For example, waits for an index block may be caused by a primary key that is based on an ascending sequence.	Examine `V$SESSION` while the problem is occurring to determine the type of block in contention.
`free buffer waits`	Buffer cache, DBWR, I/O	Slow DBWR (possibly due to I/O?) Cache too small	Examine write time using operating system statistics. Check buffer cache statistics for evidence of too small cache.
`db file scattered read`	I/O, SQL statement tuning	Poorly tuned SQL Slow I/O system	Investigate `V$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$FILESTAT` for poor read time.
`enqueue` waits (waits starting with `enq:`)	Locks	Depends on type of enqueue	Look at `V$ENQUEUE_STAT`.
library cache latch waits: `library cache`, `library cache pin`, and `library cache lock`	Latch contention	SQL parsing or sharing	Check `V$SQLAREA` to see whether there are SQL statements with a relatively high number of parse calls or a high number of child cursors (column `VERSION_COUNT`). Check parse statistics in `V$SYSSTAT` and their corresponding rate for each second.
`log buffer space`	Log buffer, I/O	Log buffer small Slow I/O system	Check the statistic `redo buffer allocation retries` in `V$SYSSTAT`. Check configuring log buffer section in configuring memory chapter. Check the disks that house the online redo logs for resource contention.
`log file sync`	I/O, over-committing	Slow disks that store the online logs Un-batched commits	Check the disks that house the online redo logs for resource contention. Check the number of transactions (`commits` + `rollbacks`) each second, from `V$SYSSTAT`.

You may also want to review the following Oracle Metalink notices on `buffer busy waits` (34405.1) and `free buffer waits` (62172.1):

- http://metalink.oracle.com/metalink/plsql/ml2_documents.showDocument?p_database_id=NOT&p_id=34405.1
- http://metalink.oracle.com/metalink/plsql/ml2_documents.showDocument?p_database_id=NOT&p_id=62172.1

You can also access these notices and related notices by searching for "busy buffer waits" and "free buffer waits" at:

`http://metalink.oracle.com`

See Also:

- "Wait Events Statistics" on page 10-17 for detailed information on each event listed in Table 10–1 and for other information to cross-check
- *Oracle Database Reference* for information about dynamic performance views

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. When Oracle commits a transaction, all modified blocks are not necessarily updated with the commit SCN immediately. In this case, it is done later on demand when the block is read or updated. This is called delayed block cleanout.

  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
  ```

  The recommended solution is 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'.

The recommended solution is to use automatic undo management.

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.

Chaining on rows larger than the block size is inevitable. You might want to consider using tablespace with larger block size for such data.

However, for smaller rows, you can avoid chaining by using sensible space parameters and good application design. For example, do *not* insert a row with key values filled in and nulls in most other columns, then update that row with the real data, causing the row to grow in size. Rather, insert rows filled with data from the start.

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 to perform poorly
- Queries that select migrated or chained rows because these must perform additional input and output

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.

Increasing PCTFREE can help 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 that have high deletion rates. If tables frequently have rows deleted, then data blocks can have partially free space in them. If rows are inserted and later expanded, then the inserted rows might land in blocks with deleted rows but still not have enough room to expand. Reorganizing the table ensures that the main free space is totally empty blocks.

Note: PCTUSED is not the opposite of PCTFREE.

See Also:

- *Oracle Database Concepts* for more information on PCTUSED
- *Oracle Database Administrator's Guide* for information on reorganizing tables

Parse-Related Statistics

The more your application parses, the more potential for 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 7, "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 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 Statistics

The `V$SESSION`, `V$SESSION_WAIT`, `V$SESSION_HISTORY`, `V$SESSION_EVENT`, and `V$SYSTEM_EVENT` views 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.

> **See Also:**
>
> - "Setting the Level of Statistics Collection" on page 10-7 for information about `STATISTICS_LEVEL` settings
> - *Oracle Database Reference* for a description of the `V$` views and the Oracle wait events

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 following views contain related, but different, views of the same data:

- `V$SESSION` lists session information for each current session. It lists either the event currently being waited for, or the event last waited for on each session. This view also contains information about blocking sessions, the wait state, and the wait time.

- V$SESSION_WAIT is a current state view. It lists either the event currently being waited for, or the event last waited for on each session, the wait state, and the wait time.
- V$SESSION_WAIT_HISTORY lists the last 10 wait events for each current session and the associated wait time.
- 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$SESSION_EVENT can show that session 124 (SID=124) had many waits on the db file scattered read, but it does not show which file and block number. However, V$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 section 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 Automatic Workload Repository. See "Overview of the Automatic Workload Repository" on page 5-8.

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:** *Oracle Database Reference* for a description of the V$SYSTEM_EVENT view

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.

Check the following V$SESSION_WAIT parameter columns:

- P1 - File ID
- P2 - Block ID
- P3 - Class ID

Causes

To determine the possible causes, first query V$SESSION to identify the value of ROW_WAIT_OBJ# when the session waits for buffer busy waits. For example:

```
SELECT row_wait_obj#
  FROM V$SESSION
 WHERE EVENT = 'buffer busy waits';
```

To identify the object and object type contended for, query DBA_OBJECTS using the value for ROW_WAIT_OBJ# that is returned from V$SESSION. For example:

```
SELECT owner, object_name, subobject_name, object_type
  FROM DBA_OBJECTS
 WHERE data_object_id = &row_wait_obj;
```

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 free list 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 (ASSM).

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 free list is a list of free data blocks that usually includes blocks existing in a number of different extents within the segment. Free lists are composed of blocks in which free space has not yet reached PCTFREE or used space has shrunk below PCTUSED. Specify the number of process free lists 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 free lists for that segment, run the following:

```
SELECT SEGMENT_NAME, FREELISTS
  FROM DBA_SEGMENTS
 WHERE SEGMENT_NAME = segment name
   AND SEGMENT_TYPE = segment type;
```

Set free lists, or increase the number of free lists. If adding more free lists does not alleviate the problem, then use free list groups (even in single instance this can make a difference). If using Oracle Real Application Clusters, then ensure that each instance has its own free list group(s).

> **See Also:** *Oracle Database Concepts* for information on automatic segment-space management, free lists, PCTFREE, and PCTUSED

data block If the contention is on tables or indexes (not the segment header):

- Check for right-hand indexes. These are indexes that are inserted into at the same point by many processes. For example, those that use sequence number generators for the key values.
- Consider using automatic segment-space management (ASSM), global hash partitioned indexes, or increasing free lists 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 scattered read to read the data into multiple discontinuous memory locations. A scattered read is usually a multiblock read. It can occur for a fast full scan (of an index) in addition to a full table scan.

The `db file scattered read` wait event identifies that a full scan is occurring. When performing a full 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 scans into the buffer cache show up as waits for 'db file scattered read'.

Check the following `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 whether there are 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.

Other things that could indicate excessive I/O load on the system include the following:

- Poor buffer cache hit ratio
- These wait 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. Gather system statistics with `DBMS_STATS` package, allowing the query optimizer to accurately cost possible access paths that use full scans
4. Use Automatic Storage Management
5. Add more disks to reduce the number of I/Os for each disk
6. Alleviate I/O hot spots by redistributing I/O across existing disks

See Also: Chapter 8, "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 toward scans)
- Lack of accurate statistics for the optimizer
- Setting the value for DB_FILE_MULTIBLOCK_READ_COUNT initialization parameter too high which favors full scans

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 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 determine, at a point in time, which sessions are waiting for I/O:

```
SELECT SQL_ADDRESS, SQL_HASH_VALUE
  FROM V$SESSION
 WHERE EVENT LIKE 'db file%read';
```

Finding the Object Requiring I/O

To determine the possible causes, first query V$SESSION to identify the value of ROW_WAIT_OBJ# when the session waits for db file scattered read. For example:

```
SELECT row_wait_obj#
  FROM V$SESSION
 WHERE EVENT = 'db file scattered read';
```

To identify the object and object type contended for, query DBA_OBJECTS using the value for ROW_WAIT_OBJ# that is returned from V$SESSION. For example:

```
SELECT owner, object_name, subobject_name, object_type
  FROM DBA_OBJECTS
 WHERE data_object_id = &row_wait_obj;
```

db file sequential read

This event signifies that the user process is reading a buffer into the SGA buffer cache and is waiting for a physical I/O call to return. A sequential read is 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'.

Check the following 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 10-20 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

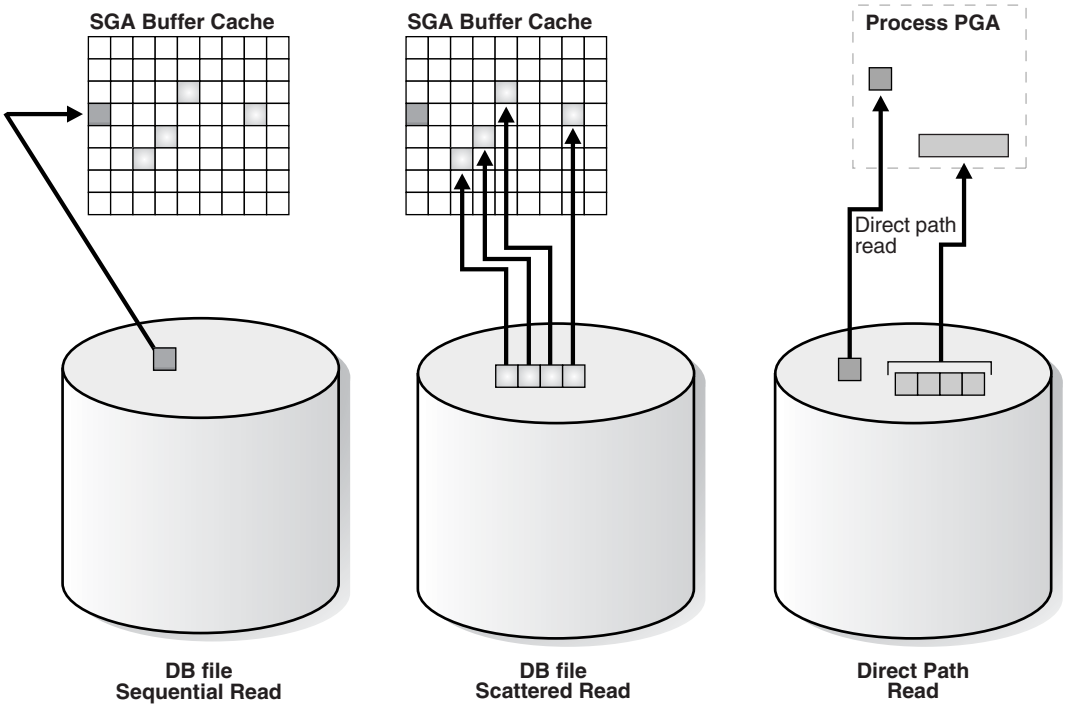
Actions

On a healthy system, physical read waits should be the biggest waits after the idle waits. However, also consider whether there are db file sequential reads on a large data warehouse that should be seeing mostly full table scans with parallel query.

Figure 10–1 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 10–1 Scattered Read, Sequential Read, and Direct Path Read



direct path read and direct path read temp

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 file sequential read`).

Check the following `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 some of the sort data is written out directly to disk. This data is later read back in, using direct reads.
- 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$TEMPSEG_USAGE` to find the SQL statement that is generating the sort. Also query the statistics from `V$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 "PGA Memory Management" on page 7-42.

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. If the full table scans are a valid part of the workload, then ensure that the I/O subsystem is configured adequately for the degree of parallelism. Consider using disk striping if you are not already using it or Automatic Storage Management (ASM).

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 10-20
- "PGA Memory Management" on page 7-42

direct path write and direct path write temp

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 `INSERT`s, 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.

See Also: *Oracle Database Administrator's Guide* for information on direct-path inserts

Check the following `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 happens in the following situations:

- Sorts are too large to fit in memory and are written to disk
- Parallel DML are issued to create/populate objects
- Direct path loads

Actions

For large sorts see "Sorts to Disk" on page 10-23.

For parallel DML, check the I/O distribution across disks and make sure that the I/O subsystem is adequately configured for the degree of parallelism.

enqueue (enq:) waits

Enqueues are locks that coordinate access to database resources. This event indicates that the session is waiting for a lock that is held by another session.

The name of the enqueue is included as part of the wait event name, in the form `enq: enqueue_type - related_details`. In some cases, the same enqueue type can be held for different purposes, such as the following related `TX` types:

- `enq: TX - allocate ITL entry`
- `enq: TX - contention`
- `enq: TX - index contention`
- `enq: TX - row lock contention`

The V$EVENT_NAME view provides a complete list of all the enq: wait events.

You can check the following V$SESSION_WAIT parameter columns for additional information:

- P1 - Lock TYPE (or name) and MODE
- P2 - Resource identifier ID1 for the lock
- P3 - Resource identifier ID2 for the lock

> **See Also:** *Oracle Database Reference* for information about Oracle enqueues

Finding Locks and Lock Holders

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. Use one of the following two queries to find the sessions holding the locks and waiting for the locks.

If there are enqueue waits, you can see these using the following statement:

```
SELECT * FROM V$LOCK WHERE request > 0;
```

To show only holders and waiters for locks being waited on, use the following:

```
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 request > 0)
   ORDER BY id1, request;
```

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 to be 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 is dictionary managed.

  > **See Also:** *Oracle 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. See *Oracle Database Administrator's Guide* for information about displaying information about space usage.
- Preallocate space in the segment, for example, by allocating extents using the ALTER TABLE ALLOCATE EXTENT SQL statement.

HW enqueue The HW enqueue is used to serialize the allocation of space beyond the high water mark of a segment.

- V$SESSION_WAIT.P2 / V$LOCK.ID1 is the tablespace number.
- V$SESSION_WAIT.P3 / V$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.

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.

TX enqueue These are acquired exclusive when a transaction initiates its first change and held until the transaction does a COMMIT or ROLLBACK.

- Waits for 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. This type of TX enqueue wait corresponds to the wait event enq: TX - row lock contention.

  The solution is to have the first session already holding the lock perform a COMMIT or ROLLBACK.

- Waits for 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. This type of TX enqueue wait corresponds to the wait event enq: TX - allocate ITL entry.

  The solution is to increase the number of ITLs available, either by changing the INITRANS or MAXTRANS for the table (either by using an ALTER statement, or by re-creating the table with the higher values).

- Waits for 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. This type of TX enqueue wait corresponds to the wait event enq: TX - row lock contention.

  The solution is to have the first session already holding the lock perform a COMMIT or ROLLBACK.

- Waits for 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. This type of TX enqueue wait corresponds to the wait event enq: TX - row lock contention.

- Waits for TX in Mode 4 can also occur waiting for a PREPARED transaction.

- Waits for TX in mode 4 also occur when a transaction inserting a row in an index has to wait for the end of an index block split being done by another transaction.

This type of TX enqueue wait corresponds to the wait event `enq: TX - index contention`.

> **See Also:** *Oracle Database Application Developer's Guide - Fundamentals* for more information about referential integrity and locking data explicitly

events in wait class other

This event belong to Other wait class and typically should not occur on a system. This event is an aggregate of all other events in the Other wait class, such as `latch free`, and is used in the V$SESSION_EVENT and V$SERVICE_EVENT views only. In these views, the events in the Other wait class will not be maintained individually in every session. Instead, these events will be rolled up into this single event to reduce the memory used for maintaining statistics on events in the Other wait class.

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 its 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 operating system 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 Chapter 8, "I/O Configuration and Design" for information on balancing I/O.

Cache is Too Small It is possible DBWR is very active because 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 "Sizing the Buffer Cache" on page 7-7.

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 and from DBWa to DBWj). 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. Multiple db writer processes are highly recommended for systems with multiple CPUs (at least one db writer for every 8 CPUs) or multiple processor groups (at least as many db writers as processor groups).

Based upon the number of CPUs and the number of processor groups, Oracle either selects an appropriate default setting for `DB_WRITER_PROCESSES` or adjusts a user-specified setting.

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`.

`DBWR_IO_SLAVES` is intended for scenarios where you cannot use multiple `DB_WRITER_PROCESSES` (for example, where you have a single CPU). 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. Asynchronous I/O at the operating system level, if you have it, is generally preferred.

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.

Idle Wait Events

These events belong to Idle wait class and 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 in Table 10–2.

Table 10–2 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	.
`pipe get`	.	X	.	.	.
`pmon timer`	X	.	.	.	.
`PX Idle Wait`	.	.	X	.	.
`PX Deq Credit: need buffer`	.	.	X	.	.
`rdbms ipc message`	X	.	.	.	.
`smon timer`	X	.	.	.	.
`SQL*Net message from client`	.	X	.	.	.
`virtual circuit status`	.	.	.	X	.

See Also: *Oracle Database Reference* for explanations of each idle wait event

latch events

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.

There is a dedicated latch-related wait event for the more popular latches that often generate significant contention. For those events, the name of the latch appears in the name of the wait event, such as `latch: library cache` or `latch: cache buffers chains`. This enables you to quickly figure out if a particular type of latch is responsible for most of the latch-related contention. Waits for all other latches are grouped in the generic `latch free` wait event.

See Also: *Oracle 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.

- 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.

Check the following `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 Waited For

```
SELECT EVENT, SUM(P3) SLEEPS, SUM(SECONDS_IN_WAIT) SECONDS_IN_WAIT
  FROM V$SESSION_WAIT
 WHERE EVENT LIKE 'latch%'
  GROUP BY EVENT;
```

A problem with the previous query is that it tells more about session tuning or instant instance tuning than instance or long-duration instance tuning.

The following query provides more information about long duration instance tuning, showing whether the latch waits are significant in the overall database time.

```
SELECT EVENT, TIME_WAITED_MICRO,
       ROUND(TIME_WAITED_MICRO*100/S.DBTIME,1) PCT_DB_TIME
  FROM V$SYSTEM_EVENT,
   (SELECT VALUE DBTIME FROM V$SYS_TIME_MODEL WHERE STAT_NAME = 'DB time') S
 WHERE EVENT LIKE 'latch%'
 ORDER BY PCT_DB_TIME ASC;
```

A more general query that is not specific to latch waits is the following:

```
SELECT EVENT, WAIT_CLASS,
      TIME_WAITED_MICRO,ROUND(TIME_WAITED_MICRO*100/S.DBTIME,1) PCT_DB_TIME
  FROM V$SYSTEM_EVENT E, V$EVENT_NAME N,
    (SELECT VALUE DBTIME FROM V$SYS_TIME_MODEL WHERE STAT_NAME = 'DB time') S
   WHERE E.EVENT_ID = N.EVENT_ID
    AND N.WAIT_CLASS NOT IN ('Idle', 'System I/O')
  ORDER BY PCT_DB_TIME ASC;
```

Table 10–3 Latch 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$SQLSTATS`) 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`
cache buffers lru chain	Buffer cache LRU lists	Excessive buffer cache throughput. For example, inefficient SQL that accesses incorrect indexes iteratively (large index range scans) or many full table scans 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	Statements with very high logical I/O or physical I/O, using unselective indexes
cache buffers chains	Buffer cache buffers	Repeated access to a block (or small number of blocks), known as a hot block	Sequence number generation code that updates a row in a table to generate the number, rather than using a sequence number generator Index leaf chasing from very many processes scanning the same unselective index with very similar predicate Identify the segment the hot block belongs to
row cache objects			

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$SQLSTATS
 WHERE EXECUTIONS < 4
 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 following example groups together statements that differ only after the first 60 bytes.

```
SELECT SUBSTR(SQL_TEXT, 1, 60), COUNT(*)
  FROM V$SQLSTATS
 WHERE EXECUTIONS < 4
 GROUP BY SUBSTR(SQL_TEXT, 1, 60)
 HAVING COUNT(*) > 1;
```

- Or report distinct SQL statements that have the same execution plan. The following query selects distinct SQL statements that share the same execution plan at least four times. These SQL statements are likely to be using literals instead of bind variables.

```
SELECT SQL_TEXT FROM V$SQLSTATS WHERE PLAN_HASH_VALUE IN
  (SELECT PLAN_HASH_VALUE
     FROM V$SQLSTATS
    GROUP BY PLAN_HASH_VALUE HAVING COUNT(*) > 4)
  ORDER BY PLAN_HASH_VALUE;
```

Reparsed Sharable SQL Check the `V$SQLSTATS` view. Enter the following query:

```
SELECT SQL_TEXT, PARSE_CALLS, EXECUTIONS
  FROM V$SQLSTATS
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:

```
SELECT pa.SID, pa.VALUE "Hard Parses", ex.VALUE "Execute Count"
  FROM V$SESSTAT pa, V$SESSTAT ex
 WHERE pa.SID = ex.SID
   AND pa.STATISTIC#=(SELECT STATISTIC#
       FROM V$STATNAME WHERE NAME = 'parse count (hard)')
   AND ex.STATISTIC#=(SELECT STATISTIC#
       FROM V$STATNAME WHERE NAME = 'execute count')
   AND pa.VALUE > 0;
```

The result is a list of all sessions and the amount of reparsing they do. For each session identifier (SID), go to `V$SESSION` to find the name of the program that causes the reparsing.

> **Note:** Because this query counts all parse calls since instance startup, it is best to look for sessions with high *rates* of parse. For example, a connection which has been up for 50 days might show a high parse figure, but a second connection might have been up for 10 minutes and be parsing at a much faster rate.

The output is similar to the following:

```
   SID  Hard Parses  Execute Count
------  -----------  -------------
     7            1             20
```

```
     8           3          12690
     6          26            325
    11          84           1619
```

cache buffers lru chain The cache buffers 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 tuning the application, bypassing the buffer cache for DSS jobs, 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 (known as a 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 X$BH table to identify the blocks protected by this latch. For example, given the address (V$LATCH_CHILDREN.ADDR) of a heavily contended latch, this queries the file and block numbers:

```
SELECT OBJ data_object_id, FILE#, DBABLK,CLASS, STATE, TCH
  FROM X$BH
 WHERE HLADDR = 'address of latch'
  ORDER BY TCH;
```

X$BH.TCH is a touch count for the buffer. A high value for X$BH.TCH indicates a hot block.

Many blocks are protected by each latch. One of these buffers will probably be the hot block. Any block with a high TCH value is a potential hot block. Perform this query a number of times, and identify the block that consistently appears in the output. After you have identified the hot block, query DBA_EXTENTS using the file number and block number, to identify the segment.

After you have identified the hot block, you can identify the segment it belongs to with the following query:

```
SELECT OBJECT_NAME, SUBOBJECT_NAME
  FROM DBA_OBJECTS
 WHERE DATA_OBJECT_ID = &obj;
```

In the query, &obj is the value of the OBJ column in the previous query on X$BH.

row cache objects The row cache objects latches protect the data dictionary.

log file parallel write

This event involves writing redo records to the redo log files from the log buffer.

library cache pin

This event manages library cache concurrency. Pinning an object causes the heaps to be loaded into memory. If a client wants to modify or examine the object, the client must acquire a pin after the lock.

library cache lock

This event controls the concurrency between clients of the library cache. It acquires a lock on the object handle so that either:

- One client can prevent other clients from accessing the same object
- The client can maintain a dependency for a long time which does not allow another client to change the object

This lock is also obtained to locate an object in the library cache.

log buffer space

This event occurs when server processes are waiting for free space in the log buffer, because all the redo is generated 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). If you have ruled out other possibilities (such as

slow disks or a full archive destination) consider increasing the number of ARCn processes. The default is 2.

- If you have mandatory remote shipped archive logs, check whether this process is slowing down because of network delays or the write is not completing because of errors.

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 Chapter 8, "I/O Configuration and Design".
- Check if there are too few, or too small redo logs. If you have a few redo logs 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 or number of redo logs. See "Sizing Redo Log Files" on page 4-4.

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 average time waited.

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 average time waited is high, then examine the session waits for the log writer and see what it is spending most of its 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.

SQL*Net Events

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. This event indicates that a server process is waiting for work from the client process. However, there are several situations where this event could accrue most of the wait time for a user experiencing poor response time. The cause could be either a network bottleneck or a resource bottleneck on the client process.

Network Bottleneck A network bottleneck can occur if the application causes a lot of traffic between server and 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.

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 message to the remote node and is waiting for a response from the database link. This time could go up because of the following:

- Network bottleneck

  For information, see "SQL*Net message from client" on page 10-36.

- Time taken to execute the SQL on the remote node

  It is useful to see the SQL 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.

- Number of round trip messages

  Each message between the session and the remote node adds latency time and processing overhead. To reduce the number of messages exchanged, use array fetches and array inserts.

SQL*Net more data to client

The server process is sending more data or messages to the client. The previous operation to the client was also a send.

> **See Also:** *Oracle Database Net Services Administrator's Guide* for a detailed discussion on network optimization

Real-Time SQL Monitoring

The real-time SQL monitoring feature of Oracle Database enables you to monitor the performance of SQL statements while they are executing. By default, SQL monitoring is automatically started when a SQL statement runs parallel, or when it has consumed at least 5 seconds of CPU or IO time in a single execution.

You can monitor the statistics for SQL statement execution using the `V$SQL_MONITOR` and `V$SQL_PLAN_MONITOR` views. These views can be used in conjunction with the following views to get additional information about the execution being monitored:

- `V$ACTIVE_SESSION_HISTORY`
- `V$SESSION`
- `V$SESSION_LONGOPS`
- `V$SQL`
- `V$SQL_PLAN`

Once monitoring is initiated, an entry is added to the dynamic performance view `V$SQL_MONITOR`. This entry tracks key performance metrics collected for the execution, including the elapsed time, CPU time, number of reads and writes, I/O wait time and various other wait times. These statistics are refreshed in near real-time as the statement executes, generally once every second. Once the execution ends, monitoring information is not deleted immediately, but is kept in the `V$SQL_`

MONITOR view for at least one minute. The entry will eventually be deleted so its space can be reclaimed as new statements are monitored.

The V$SQL_MONITOR view contains a subset of the statistics available in V$SQL. However, unlike V$SQL, monitoring statistics are not cumulative over several executions. Instead, one entry in V$SQL_MONITOR is dedicated to a single execution of a SQL statement. If two executions of the same SQL statement are being monitored, each of these executions will have a separate entry in V$SQL_MONITOR.

To uniquely identify two executions of the same SQL statement, a composite key called an execution key is generated. This execution key is composed of three attributes, each corresponding to a column in V$SQL_MONITOR:

- SQL identifier to identify the SQL statement (SQL_ID)
- Start execution timestamp (SQL_EXEC_START)
- An internally generated identifier to ensure that this primary key is truly unique (SQL_EXEC_ID)

This section contains the following topics:

SQL Plan Monitoring

Real-time SQL monitoring also includes monitoring statistics for each operation in the execution plan of the SQL statement being monitored. This data is visible in the V$SQL_PLAN_MONITOR view. Similar to the V$SQL_MONITOR view, statistics in V$SQL_PLAN_MONITOR are updated every second as the SQL statement is being executed. These statistics persist after the execution ends, with the same duration as V$SQL_MONITOR. There will be multiple entries in V$SQL_PLAN_MONITOR for every SQL statement being monitored; each entry will correspond to an operation in the execution plan of the statement.

Parallel Execution Monitoring

Parallel queries, DML and DDL statements are automatically monitored as soon as execution begins. Monitoring information for each process participating in the parallel execution is recorded as separate entries in the V$SQL_MONITOR and V$SQL_PLAN_MONITOR views. As a result, the V$SQL_MONITOR view will have one entry for the parallel execution coordinator process, and one entry for each parallel execution server process. Each of these entries will have corresponding entries in the V$SQL_PLAN_MONITOR view. Since the processes allocated for the parallel execution of a SQL statement are cooperating for the same execution, these entries share the same execution key (the composite SQL_ID, SQL_EXEC_START and SQL_EXEC_ID). You can therefore aggregate the execution key to determine the overall statistics for a parallel execution.

Generating the SQL Monitor Report

You can use the SQL monitor report to view SQL monitoring data. The SQL monitor report uses data from several views, including:

- GV$SQL_MONITOR

- GV$SQL_PLAN_MONITOR
- GV$SQL
- GV$SQL_PLAN
- GV$ACTIVE_SESSION_HISTORY
- GV$SESSION_LONGOPS

To generate the SQL monitor report, run the REPORT_SQL_MONITOR function in the DBMS_SQLTUNE package:

```
variable my_rept CLOB;
BEGIN
  :my_rept :=DBMS_SQLTUNE.REPORT_SQL_MONITOR();
END;
/

print :my_rept
```

The DBMS_SQLTUNE.REPORT_SQL_MONITOR function accepts several input parameters to specify the execution, the level of detail in the report, and the report type ('TEXT', 'HTML', or 'XML'). By default, a text report is generated for the last execution that was monitored if no parameters are specified as shown in the example.

> **See Also:** *Oracle Database PL/SQL Packages and Types Reference* for information about the DBMS_SQLTUNE package

Example 10–1 shows the output of the SQL Monitor Report for the last execution of a SQL statement that was monitored.

Example 10–1 Sample SQL Monitor Report

```
set long 10000000
set longchunksize 10000000
set linesize 200
select dbms_sqltune.report_sql_monitor from dual;

SQL Text
------------------------------------------------------------------------------------------
select * from (select O_ORDERDATE, sum(O_TOTALPRICE)
               from  orders o, lineitem l
               where l.l_orderkey = o.o_orderkey
               group by o_orderdate
               order by o_orderdate) where rownum < 100
------------------------------------------------------------------------------------------

Global Information
 Status              :  EXECUTING
 Instance ID         :  1
 Session ID          :  980
 SQL ID              :  br4m75c20p97h
 SQL Execution ID    :  16777219
 Plan Hash Value     :  2992965678
 Execution Started   :  06/07/2007 08:36:42
 First Refresh Time  :  06/07/2007 08:36:46
 Last Refresh Time   :  06/07/2007 08:40:02

--------------------------------------------------------------------------------
| Elapsed |   Cpu   |    IO    | Application |  Other   | Buffer | Reads | Writes |
| Time(s) | Time(s) | Waits(s) |  Waits(s)   | Waits(s) |  Gets  |       |        |
```

```
--------------------------------------------------------------------------------
|     198 |     140 |       56 |        0.31 |     1.44 |  1195K | 1264K |  84630 |
--------------------------------------------------------------------------------

SQL Plan Monitoring Details
================================================================================================
| Id   |         Operation          |   Name    |  Rows   | Cost  |   Time    | Start  |
|      |                            |           | (Estim) |       | Active(s) | Active |
================================================================================================
|    0 | SELECT STATEMENT           |           |         |  125K |           |        |
|    1 |   COUNT STOPKEY            |           |         |       |           |        |
|    2 |    VIEW                    |           |    2406 |  125K |           |        |
|    3 |     SORT GROUP BY STOPKEY  |           |    2406 |  125K |        99 |   +101 |
| -> 4 |      HASH JOIN             |           |    8984K|  123K |       189 |    +12 |
|      |                            |           |         |       |           |        |
|    5 |       INDEX FAST FULL SCAN | I_L_OKEY  |    8984K| 63191 |        82 |     +1 |
|      |                            |           |         |       |           |        |
|    6 |       PARTITION RANGE ALL  |           |  44913K | 57676 |        94 |    +84 |
|    7 |        PARTITION HASH ALL  |           |  44913K | 57676 |        94 |    +84 |
|    8 |         TABLE ACCESS FULL  | ORDERS    |  44913K | 57676 |        95 |    +84 |
|      |                            |           |         |       |           |        |
|      |                            |           |         |       |           |        |
================================================================================================

continuation of above table
================================================================================================
 Starts |   Rows    | Memory | Temp | Activity  |      Activity Detail      | Progress |
        | (Actual)  |        |      | (percent) |        (sample #)         |          |
================================================================================================
      1 |           |        |      |           |                           |          |
      1 |           |        |      |           |                           |          |
      1 |           |        |      |           |                           |          |
      1 |         0 |        |      |      4.02 | Cpu (8)                   |          |
      1 |   28130K  | 10000K | 724M |     25.13 | Cpu (48)                  | 87%      |
        |           |        |      |           | direct path read temp (2) |          |
      1 |   32734K  |        |      |     34.17 | Cpu (58)                  | 100%     |
        |           |        |      |           | direct path read (10)     |          |
      1 |   45000K  |        |      |           |                           |          |
     84 |   45000K  |        |      |           |                           |          |
    672 |   45000K  |        |      |     36.68 | Cpu (28)                  |          |
        |           |        |      |           | reliable message (3)      |          |
        |           |        |      |           | direct path read (42)     |          |
================================================================================================
```

In the Global Information section of this report, the Status field shows that this statement is still executing. The Time Active(s) column shows how long the operation has been active (the delta in seconds between the first and the last active time). The Start Active column shows, in seconds, when the operation in the execution plan started relative to the SQL statement execution start time. In this report, the fast full scan operation at Id 5 was the first to start (+1s Start Active) and ran for the first 82 seconds.

The Starts column shows the number of times the operation in the execution plan was executed. The Rows (Actual) column indicates the number of rows produced, and the Rows (Estim) column shows the estimated cardinality from the optimizer. The Memory and Temp columns indicate the amount of memory and temporary space consumed by each operation of the execution plan.

The Activity (percent) and Activity Detail (sample #) columns are derived by joining the `V$SQL_PLAN_MONITOR` and `V$ACTIVE_SESSION_HISTORY` views. Activity

(percent) shows the percentage of database time consumed by each operation of the execution plan. Activity Detail (sample#) shows the nature of that activity (such as CPU or wait event). In this report, this column shows that most of the database time, 36.68%, is consumed by operation Id 8 (TABLE ACCESS FULL of ORDERS). This activity consists of 73 samples (28+3+42), of which more than half of the activity is attributed to direct path read (42 samples), and a third to CPU (28 samples).

The last column, Progress, shows progress monitoring information for the operation from the V$SESSION_LONGOPS view. In this report, it shows that the hash-join operation is 87% complete.

Enabling and Disabling SQL Monitoring

The SQL monitoring feature is enabled by default when the STATISTICS_LEVEL initialization parameter is either set to ALL or TYPICAL (the default value). Additionally, the CONTROL_MANAGEMENT_PACK_ACCESS parameter must be set to DIAGNOSTIC+TUNING (the default value) because SQL monitoring is a feature of the Oracle Database Tuning Pack. SQL monitoring starts automatically for all long running queries.

Two statement-level hints are available to force or prevent a SQL statement from being monitored. To force SQL monitoring, use the MONITOR hint:

```
select /*+MONITOR*/ from dual;
```

This hint is effective only when the CONTROL_MANAGEMENT_PACK_ACCESS parameter is set to DIAGNOSTIC+TUNING. To prevent the hinted SQL statement from being monitored, use the NO_MONITOR reverse hint.

> **See Also:** *Oracle Database SQL Reference* for information about using the MONITOR and NO_MONITOR hints

Part IV

Optimizing SQL Statements

Part IV provides information on understanding and managing your SQL statements for optimal performance and discusses Oracle SQL-related performance tools.

The chapters in this part include:

11

The Query Optimizer

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

The chapter contains the following sections:

Optimizer Operations

A SQL statement can be executed in many different ways, such as full table scans, index scans, nested loops, and hash joins. The query optimizer determines the most efficient way to execute a SQL statement after considering many factors related to the objects referenced and the conditions specified in the query. This determination is an important step in the processing of any SQL statement and can greatly affect execution time.

> **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, because better information is available.

The output from the optimizer is a plan that describes an optimum method of execution. The Oracle server provides query optimization.

For any SQL statement processed by Oracle, the optimizer performs the operations listed in Table 11–1.

Table 11–1 Optimizer Operations

Operation	Description
Evaluation of expressions and conditions	The optimizer first evaluates expressions and conditions containing constants as fully as possible.

Table 11–1 (Cont.) Optimizer Operations

Operation	Description
Statement transformation	For complex statements involving, for example, correlated subqueries or views, the optimizer might transform the original statement into an equivalent join statement.
Choice of optimizer goals	The optimizer determines the goal of optimization. See "Choosing an Optimizer Goal" on page 11-2.
Choice of access paths	For each table accessed by the statement, the optimizer chooses one or more of the available access paths to obtain table data. See "Understanding Access Paths for the Query Optimizer" on page 11-14.
Choice of join orders	For a join statement that joins more than two tables, the optimizer chooses which pair of tables is joined first, and then which table is joined to the result, and so on. See "How the Query Optimizer Chooses Execution Plans for Joins" on page 11-24.

You can influence the optimizer's choices by setting the optimizer goal, and by gathering representative statistics for the query optimizer. The optimizer goal is either throughput or response time. See "Choosing an Optimizer Goal" on page 11-2 and "Query Optimizer Statistics in the Data Dictionary" on page 11-4.

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 instruct the optimizer about how a statement should be executed.

See Also:

- *Oracle Database Concepts* for an overview of SQL processing and the optimizer
- *Oracle Database Data Cartridge Developer's Guide* for information about the extensible optimizer
- "Choosing an Optimizer Goal" on page 11-2 for more information on optimization goals
- Chapter 13, "Managing Optimizer Statistics" for information on gathering and using statistics
- Chapter 19, "Using Optimizer Hints" for more information about using hints in SQL statements

Choosing an Optimizer Goal

By default, the goal of the query optimizer is the best throughput. This means that it chooses the least amount of resources necessary to process all rows accessed by the statement. Oracle can also optimize a statement with the goal of best response time. This means that it uses the least amount of resources necessary to process the first row accessed by a SQL statement.

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 SQL Hints for Changing the Query Optimizer Goal
- Query Optimizer Statistics in the Data Dictionary

OPTIMIZER_MODE Initialization Parameter

The `OPTIMIZER_MODE` initialization parameter establishes the default behavior for choosing an optimization approach for the instance. The possible values and description are listed in Table 11–2.

Table 11–2 OPTIMIZER_MODE Initialization Parameter Values

Value	Description
`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). This is the default value.
`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 the first *n* number of rows; *n* can equal 1, 10, 100, or 1000.
`FIRST_ROWS`	The optimizer uses a mix of cost and heuristics to find a best plan for fast delivery of the first few rows. Note: Using heuristics sometimes leads the query optimizer to generate a plan with a cost that is significantly larger than the cost of a plan without applying the heuristic. `FIRST_ROWS` is available for backward compatibility and plan stability; use `FIRST_ROWS_`*n* instead.

You can change the goal of the query optimizer for all SQL statements in a session by changing the parameter value in initialization file or by the `ALTER SESSION SET OPTIMIZER_MODE` statement. For example:

- The following statement in an initialization parameter file establishes the goal of the query optimizer for all sessions of the instance to best response time:

```
OPTIMIZER_MODE = FIRST_ROWS_1
```

- The following SQL statement changes the goal of the query optimizer for the current session to best response time:

```
ALTER SESSION SET OPTIMIZER_MODE = FIRST_ROWS_1;
```

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 SQL Hints for Changing the Query Optimizer Goal

To specify the goal of the query optimizer for an individual SQL statement, use one of the hints in Table 11–3. Any of these hints in an individual SQL statement can override the OPTIMIZER_MODE initialization parameter for that SQL statement.

Table 11–3 Hints for Changing the Query Optimizer Goal

Hint	Description
FIRST_ROWS(n)	This hint instructs Oracle to optimize an individual SQL statement with a goal of best response time to return the first n number of rows, where n equals any positive integer. The hint uses a cost-based approach for the SQL statement, regardless of the presence of statistic.
ALL_ROWS	This hint explicitly chooses the cost-based approach to optimize a SQL statement with a goal of best throughput.

See Also: Chapter 19, "Using Optimizer Hints" for information on how to use hints

Query Optimizer Statistics in the Data Dictionary

The statistics used by the query optimizer 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.

To maintain the effectiveness of the query optimizer, you must have statistics that are representative of the data. For table columns that contain values with large variations in number of duplicates, called skewed data, you should collect histograms.

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

If no statistics are available when using query optimization, the optimizer will do dynamic sampling depending on the setting of the OPTMIZER_DYNAMIC_SAMPLING initialization parameter. This may cause slower parse times so for best performance, the optimizer should have representative optimizer statistics.

See Also:

- Chapter 13, "Managing Optimizer Statistics"
- "Viewing Histograms" on page 13-23 for a description of histograms
- "Estimating Statistics with Dynamic Sampling" on page 13-20

Enabling and Controlling Query Optimizer Features

This section contains some of the initialization parameters specific to the optimizer. The following sections are especially useful when tuning Oracle applications.

See Also: *Oracle Database Reference* for information about initialization parameters

Enabling Query Optimizer Features

You enable optimizer features by setting the OPTIMIZER_FEATURES_ENABLE initialization parameter.

OPTIMIZER_FEATURES_ENABLE Parameter

The OPTIMIZER_FEATURES_ENABLE parameter acts as an umbrella parameter for the query optimizer. This parameter can be used to enable a series of optimizer-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, 8.1.7, and 9.2.0. For example, the following setting enables the use of the optimizer features in generating query plans in Oracle 10*g*, Release 1.

```
OPTIMIZER_FEATURES_ENABLE=10.0.0;
```

The OPTIMIZER_FEATURES_ENABLE parameter was introduced with the main goal to allow customers to upgrade the Oracle server, yet preserve the old behavior of the query optimizer 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 upgrade results in the query 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 because of new optimizer features in a new release. In such a case, you can set the OPTIMIZER_FEATURES_ENABLE parameter to an earlier version. For example, to preserve the behavior of the query optimizer to release 8.1.5, set the parameter as follows:

```
OPTIMIZER_FEATURES_ENABLE=8.1.5;
```

This statement disables all new optimizer features that were added in releases following release 8.1.5. If you upgrade to a new release and you want to enable the features available with that release, then you do not need to explicitly set the OPTIMIZER_FEATURES_ENABLE initialization parameter

---

Note: Oracle does not recommend explicitly setting the OPTIMIZER_FEATURES_ENABLE parameter to an earlier release. To avoid possible SQL performance regression that may result from execution plan changes, consider using SQL plan management instead. For more information, see Chapter 15, "Using SQL Plan Management"

See Also: *Oracle Database Reference* for information about optimizer features that are enabled when you set the OPTIMIZER_FEATURES_ENABLE parameter to each of the release values

Controlling the Behavior of the Query Optimizer

This section lists some initialization parameters that can be used to control the behavior of the query optimizer. 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. Converting the values 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. If this parameter is not set explicitly (or is set is 0), the optimizer will use a default value of 8 when costing full table scans and index fast full scans.

OPTIMIZER_INDEX_CACHING

This parameter controls the costing of an index probe in conjunction with a nested loop. The range of values `0` to `100` for `OPTIMIZER_INDEX_CACHING` indicates percentage of index blocks in the buffer cache, which modifies the optimizer's assumptions about index caching for nested loops and IN-list iterators. A value of `100` infers that 100% of the index blocks are likely to be found in the buffer cache and the optimizer adjusts the cost of an index probe or nested loop accordingly. Use caution 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` to `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 one-tenth the normal cost of an index access path.

OPTIMIZER_MODE

This initialization parameter sets the mode of the optimizer at instance startup. The possible values are `ALL_ROWS`, `FIRST_ROWS_`*n*, and `FIRST_ROWS`. For descriptions of these parameter values, see "OPTIMIZER_MODE Initialization Parameter" on page 11-3.

PGA_AGGREGATE_TARGET

This parameter automatically controls the amount of memory allocated for sorts and hash joins. Larger amounts of memory allocated for sorts or hash joins reduce the optimizer cost of these operations. See "PGA Memory Management" on page 7-42.

STAR_TRANSFORMATION_ENABLED

This parameter, if set to `true`, enables the query optimizer to cost a star transformation for star queries. The star transformation combines the bitmap indexes on the various fact table columns.

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

Understanding the Query Optimizer

The query optimizer 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 query optimizer

also considers hints, which are optimization instructions placed in a comment in the statement.

> **See Also:** Chapter 19, "Using Optimizer Hints" for detailed information on hints

The query optimizer performs the following steps:

1. The optimizer generates a set of potential plans for the SQL statement based on 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, which includes 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.

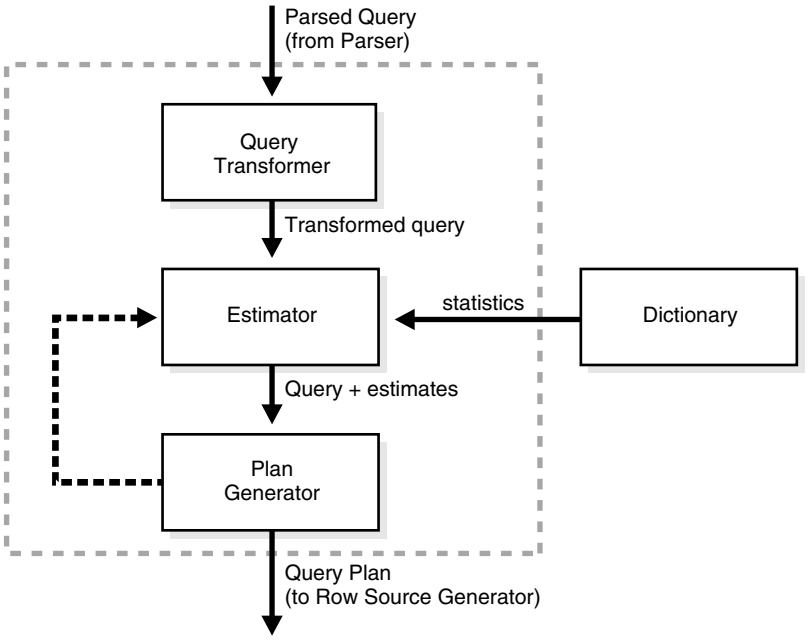
Components of the Query Optimizer

The query optimizer operations include:

- Transforming Queries
- Estimating
- Generating Plans

Query optimizer components are illustrated in Figure 11–1.

Figure 11–1 Query Optimizer Components



Transforming Queries

The input to the 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. Several different query transformation techniques are employed by the query transformer, including:

- View Merging
- Predicate Pushing
- Subquery Unnesting
- Query Rewrite with Materialized Views

Any combination of these transformations can 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 analyze the view query block separately and generate a view subplan. The optimizer then processes the rest of the query by using the view subplan in the generation of an overall query plan. This technique usually leads to a suboptimal query plan, because the view is optimized separately from rest of the query.

The query transformer then removes the potentially 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. Generating a subplan is no longer necessary, because the 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.

This technique improves the subplan of the non-merged view, because the pushed-in predicates can be used either to access indexes or to act as filters.

Subquery Unnesting Often the performance of queries that contain subqueries can be improved 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 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 with a result that 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. This technique 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 a lower cost than the plan generated with the materialized views.

Peeking of User-Defined Bind Variables

The query optimizer peeks at the values of user-defined bind variables on the first invocation of a cursor. This feature enables the optimizer to determine the selectivity of any `WHERE` clause condition as if literals have been used instead of bind variables.

To ensure the optimal choice of cursor for a given bind value, Oracle Database uses bind-aware cursor matching. The system monitors the data access performed by the query over time, depending on the bind values. If bind peeking takes place, and a histogram is used to compute selectivity of the predicate containing the bind variable, then the cursor is marked as a bind-sensitive cursor. Whenever a cursor is determined to produce significantly different data access patterns depending on the bind values, that cursor is marked as bind-aware, and Oracle Database will switch to bind-aware cursor matching to select the cursor for that statement. When bind-aware cursor matching is enabled, plans are selected based on the bind value and the optimizer's estimate of its selectivity. With bind-aware cursor matching, it is possible that a SQL statement with user-defined bind variable will have multiple execution plans, depending on the bind values.

When bind variables are used in a SQL statement, it is assumed that cursor sharing is intended and that different invocations will use the same execution plan. If different invocations of the cursor will significantly benefit from different execution plans, then bind-aware cursor matching is required. Bind peeking does not work for all clients, but a specific set of clients.

Consider the following example:

```
SELECT avg(e.salary), d.department_name
    FROM employees e, departments d
    WHERE e.job_id = :job
       AND e.department_id = d.department_id
       GROUP BY d.department_name;
```

In this example, the column `job_id` is skewed because there are a lot more Sales Representatives (`job_id = 'SA_REP'`) than there are Vice Presidents (`job_id = 'AD_VP'`). Therefore, the best plan for this query depends on the value of the bind variable. In this case, it is more efficient to use an index when the `job_id` is `AD_VP`, and a full table scan when the `job_id` is `SA_REP`. The optimizer will peek at the first

value (`'AD_VP'`) and choose an index, and the cursor will be marked as a bind-sensitive cursor. If the next time the query is executed and the bind value is `MK_REP` (Marketing Representative) and this bind value has low selectivity, the optimizer may decide to mark the cursor as bind-aware and hard parse the statement to generate a new plan that performs a full table scan.

The selectivity ranges, cursor information (such as whether a cursor is bind-aware or bind-sensitive), and execution statistics are available using the V$ views for extended cursor sharing. The `V$SQL_CS_STATISTICS` view contains execution statistics for each cursor, and can be used for performance tuning by comparing the cursor executions generated with different bind sets.

> **See Also:** *Oracle Database Data Warehousing Guide* for more information about query rewrite

Estimating

The estimator generates three different types of measures:

- Selectivity
- Cardinality
- 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 measure, selectivity, 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 a 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. Selectivity lies in a value range from 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.

If no statistics are available then the optimizer either uses dynamic sampling or an internal default value, depending on the value of the `OPTIMIZER_DYNAMIC_SAMPLING` initialization parameter. 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. See "Estimating Statistics with Dynamic Sampling" on page 13-20.

When statistics are available, the estimator uses them to estimate selectivity. For example, for an equality predicate (`last_name = 'Smith'`), selectivity is set to the reciprocal of the number *n* 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. The histogram captures the distribution of different values in a column, so it yields better selectivity estimates. Having histograms on columns that contain skewed data (in other words, values with large variations in number of duplicates) greatly helps the query optimizer generate good selectivity estimates.

> **See Also:** "Viewing Histograms" on page 13-23 for a description of histograms

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.

Cost The cost represents units of work or resource used. The query optimizer uses disk I/O, CPU usage, and memory usage as units of work. So, the cost used by the query optimizer 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 by 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** determines the number of units of work required to get 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 be scanned and the multiblock read count value. The cost of an index scan depends on the levels in the B-tree, the number of index leaf blocks to be scanned, and the number of rows to be fetched using the rowid in the index keys. The cost of fetching rows using rowids depends on the index clustering factor. See "Assessing I/O for Blocks, not Rows" on page 11-17.

The **join cost** represents the combination of the individual access costs of the two row sets being joined, plus the cost of the join operation.

> **See Also:** "Understanding Joins" on page 11-23 for more information on joins

Generating Plans

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 `table1`, `table2`, and `table3`, table `table1` is accessed first. Next, `table2` is accessed, and its data is joined to `table1` data to produce a join of `table1` and `table2`. Finally, `table3` is accessed, and its data is joined to the result of the join between `table1` and `table2`.

The plan for a query is established by first generating subplans for each of the nested subqueries and unmerged views. Each nested subquery or unmerged 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 various 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.

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 the 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 the current best cost is small, then the plan generator ends the search swiftly, because further cost improvement will not be significant.

The cutoff works 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.

Reading and Understanding Execution Plans

To execute a SQL 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 Query Optimizer" on page 11-14
- Chapter 12, "Using EXPLAIN PLAN"

Overview of EXPLAIN PLAN

You can examine the execution plan chosen by the optimizer for a SQL statement by using the EXPLAIN PLAN statement. When the statement is issued, the optimizer chooses an execution plan and then inserts data describing the plan into a database table. Simply issue the EXPLAIN PLAN statement and then query the output table.

These are the basics of using the EXPLAIN PLAN statement:

- Use the SQL script `UTLXPLAN.SQL` to create a sample output table called `PLAN_TABLE` in your schema. See "The PLAN_TABLE Output Table" on page 12-4.
- Include the EXPLAIN PLAN FOR clause prior to the SQL statement. See "Running EXPLAIN PLAN" on page 12-5.
- After issuing the EXPLAIN PLAN statement, use one of the scripts or package provided by Oracle to display the most recent plan table output. See "Displaying PLAN_TABLE Output" on page 12-6.
- The execution order in EXPLAIN PLAN output 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 normally executed first.

Notes:

- The `EXPLAIN PLAN` output tables in this chapter were displayed with the `utlxpls.sql` script.
- The steps in the `EXPLAIN PLAN` output in this chapter may be different on your system. The optimizer may choose different execution plans, depending on database configurations.

Example 11–1 uses EXPLAIN PLAN to examine a SQL statement that selects the `employee_id`, `job_title`, `salary`, and `department_name` for the employees whose IDs are less than 103.

Example 11–1 Using EXPLAIN PLAN

```
EXPLAIN PLAN FOR
SELECT e.employee_id, j.job_title, e.salary, d.department_name
    FROM employees e, jobs j, departments d
    WHERE  e.employee_id < 103
       AND e.job_id = j.job_id
       AND e.department_id = d.department_id;
```

The resulting output table in Example 11–2 shows the execution plan chosen by the optimizer to execute the SQL statement in the example:

Example 11–2 EXPLAIN PLAN Output

```
-------------------------------------------------------------------------------------
| Id  | Operation                     |  Name        | Rows  | Bytes | Cost (%CPU)|
-------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |              |     3 |   189 |    10  (10)|
|   1 |  NESTED LOOPS                 |              |     3 |   189 |    10  (10)|
|   2 |   NESTED LOOPS                |              |     3 |   141 |     7  (15)|
|*  3 |    TABLE ACCESS FULL          | EMPLOYEES    |     3 |    60 |     4  (25)|
|   4 |    TABLE ACCESS BY INDEX ROWID| JOBS         |    19 |   513 |     2  (50)|
|*  5 |     INDEX UNIQUE SCAN         | JOB_ID_PK    |     1 |       |            |
|   6 |   TABLE ACCESS BY INDEX ROWID | DEPARTMENTS  |    27 |   432 |     2  (50)|
|*  7 |    INDEX UNIQUE SCAN          | DEPT_ID_PK   |     1 |       |            |
-------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   3 - filter("E"."EMPLOYEE_ID"<103)
   5 - access("E"."JOB_ID"="J"."JOB_ID")
   7 - access("E"."DEPARTMENT_ID"="D"."DEPARTMENT_ID")
```

Steps in the Execution Plan

Each row in the output table corresponds to a single step in the execution plan. Note that the step Ids with asterisks are listed in the Predicate Information section.

> **See Also:** Chapter 12, "Using EXPLAIN PLAN"

Each step of the execution plan returns a set of rows that either is used by the next step or, in the last step, is returned to the user or application issuing the SQL statement. A set of rows returned by a step is called a row set.

The numbering of the step Ids reflects the order in which they are displayed in response to the `EXPLAIN PLAN` statement. Each step of the execution plan either retrieves rows from the database or accepts rows from one or more row sources as input.

- The following steps in Example 11–2 physically retrieve data from an object in the database:
  - Step 3 reads all rows of the `employees` table.
  - Step 5 looks up each `job_id` in `JOB_ID_PK` index and finds the rowids of the associated rows in the `jobs` table.
  - Step 4 retrieves the rows with rowids that were returned by Step 5 from the `jobs` table.
  - Step 7 looks up each `department_id` in `DEPT_ID_PK` index and finds the rowids of the associated rows in the `departments` table.

- Step 6 retrieves the rows with rowids that were returned by Step 7 from the `departments` table.

- The following steps in Example 11–2 operate on rows returned by the previous row source:
  - Step 2 performs the nested loop operation on `job_id` in the `jobs` and `employees` tables, 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 2.
  - Step 1 performs the nested loop operation, accepting row sources from Step 2 and Step 6, joining each row from Step 2 source to its corresponding row in Step 6, and returning the resulting rows to Step 1.

See Also:

- "Understanding Access Paths for the Query Optimizer" on page 11-14 for more information on access paths
- "Understanding Joins" on page 11-23 for more information on the methods by which Oracle joins row sources

Understanding Access Paths for the Query Optimizer

Access paths are ways in which data is retrieved from the database. In general, index access paths should be used for statements that retrieve a small subset of table rows, while full scans are more efficient when accessing a large portion of the table. Online transaction processing (OLTP) applications, 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, tend to use partitioned tables and perform full scans of the relevant partitions.

This section describes the data access paths that can be used to locate and retrieve any row in any table.

- Full Table Scans
- Rowid Scans
- Index Scans
- Cluster Access
- Hash Access
- Sample Table Scans
- How the Query Optimizer Chooses an Access Path

Full Table Scans

This type of scan reads all rows from a table and filters out those that do not meet the selection criteria. During a full table scan, all blocks in the table that are under the high water mark are scanned. The high water mark indicates the amount of used space, or space that had been formatted to receive data. Each row is examined to determine whether it satisfies the statement's `WHERE` clause.

When Oracle performs a full table 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.

Example 11–2, "EXPLAIN PLAN Output" on page 11-13 contains an example of a full table scan on the `employees` table.

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 a table. This is because full table scans can use larger I/O calls, and making fewer large I/O calls is cheaper than making many smaller calls.

When the Optimizer Uses Full Table Scans

The optimizer uses a full table scan in any of the following cases:

Lack of Index If the query is unable to use any existing indexes, then it uses a full table scan. For example, if there is a function used on the indexed column in the query, the optimizer is unable to use the index and instead uses a full table scan.

If you need to use the index for case-independent searches, then either do not permit mixed-case data in the search columns or create a function-based index, such as `UPPER(last_name)`, on the search column. See "Using Function-based Indexes for Performance" on page 14-7.

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

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

High Degree of Parallelism A high degree of parallelism for a table skews the optimizer toward 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`*`)` to instruct the optimizer to use a full table scan. For more information on the `FULL` hint, see "Hints for Access Paths" on page 19-3.

You can use the `CACHE` and `NOCACHE` hints to indicate where the retrieved blocks are placed in the buffer cache. The `CACHE` hint instructs the optimizer to place the retrieved blocks at the most recently used end of the LRU list in the buffer cache when a full table scan is performed.

Small tables are automatically cached according to the criteria in Table 11–4.

Table 11–4 Table Caching Criteria

Table Size	Size Criteria	Caching
Small	Number of blocks < 20 or 2% of total cached blocks, whichever is larger	If `STATISTICS_LEVEL` is se to `TYPICAL` or higher, Oracle decides whether to cache a table depending on the table scan history. The table is cached only if a future table scan is likely to find the cached blocks. If `STATISTICS_LEVEL` is set to `BASIC`, the table is not cached.

Table 11–4 (Cont.) Table Caching Criteria

Table Size	Size Criteria	Caching
Medium	Larger than a small table, but < 10% of total cached blocks	Oracle decides whether to cache a table based on its table scan and workload history. It caches the table only if a future table scan is likely to find the cached blocks.
Large	> 10% of total cached blocks	Not cached

Automatic caching of small tables is disabled for tables that are created or altered with the `CACHE` attribute.

Parallel Query Execution

When a full table scan is required, response time can be improved by using multiple parallel execution servers for scanning the table. Parallel queries are used generally in low-concurrency data warehousing environments, because of the potential resource usage.

See Also: *Oracle Database Data Warehousing Guide*

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 specifying its rowid is the fastest way to retrieve a single row, because the exact location of the row in the database is specified.

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.

In Example 11–2, "EXPLAIN PLAN Output" on page 11-13, an index scan is performed the `jobs` and `departments` tables. The rowids retrieved are used to return the row data.

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 occur.

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 *Oracle Database Application Developer's Guide - Fundamentals*.

Index Scans

In this method, a row is retrieved by traversing the index, using the indexed column values specified by the statement. An index scan retrieves data from an index based on the value of one or more columns in 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 by using either a table access by rowid or a cluster scan.

An index scan can be one of the following types:

- Assessing I/O for Blocks, not Rows
- Index Unique Scans
- Index Range Scans
- Index Range Scans Descending
- Index Skip Scans
- Full Scans
- Fast Full Index Scans
- Index Joins
- Bitmap Indexes

Assessing I/O for Blocks, not 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 called 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. Consequently, 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.

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 use a range scan to fetch rows by rowid, because more blocks in the table need to be visited to return the data. Example 11–3 shows how the clustering factor can affect cost.

Example 11–3 Effects of Clustering Factor on Cost

Assume the following situation:

- There is a table with 9 rows.
- There is a non-unique index on `col1` for table.
- The `c1` column currently stores the values `A`, `B`, and `C`.
- The table only has three Oracle blocks.

Case 1: The index clustering factor is low for the rows as they are arranged in the following diagram.

```
Block 1       Block 2       Block 3
-------       -------       --------
```

```
A  A  A      B  B  B      C  C  C
```

This is because the rows that have the same indexed column values for `c1` are located within the same physical blocks in the table. The cost of using a range scan to return all of the rows that have the value `A` is low, because only one block in the table needs to be read.

Case 2: If the same rows in the table are rearranged so that the index values are scattered across the table blocks (rather than collocated), then the index clustering factor is higher.

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

This is because all three blocks in the table must be read in order to retrieve all rows with the value `A` in `col1`.

Index Unique Scans

This scan returns, at most, a single rowid. Oracle performs a unique scan if a statement contains a `UNIQUE` or a `PRIMARY KEY` constraint that guarantees that only a single row is accessed.

In Example 11–2, "EXPLAIN PLAN Output" on page 11-13, an index scan is performed on the `jobs` and `departments` tables, using the `job_id_pk` and `dept_id_pk` indexes respectively.

When the Optimizer Uses Index Unique Scans This access path is used when all columns of a unique (B-tree) index or an index created as a result of a primary key constraint are specified with equality conditions.

> **See Also:** *Oracle 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 use a hint to do a unique scan. There might be cases where the table is across a database link and being accessed 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 an access path (range scan or unique scan). For more information on the `INDEX` hint, see "Hints for Access Paths" on page 19-3.

Index Range Scans

An 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 rowid.

If data must 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 option and avoids a sort.

In Example 11–4, the order has been imported from a legacy system, and you are querying the order by the reference used in the legacy system. Assume this reference is the `order_date`.

Example 11–4 Index Range Scan

```
SELECT order_status, order_id
  FROM orders
 WHERE order_date = :b1;

-----------------------------------------------------------------------------------------
| Id  | Operation                   |  Name              | Rows  | Bytes | Cost (%CPU)|
-----------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT            |                    |     1 |    20 |     3  (34)|
|   1 |  TABLE ACCESS BY INDEX ROWID| ORDERS             |     1 |    20 |     3  (34)|
|*  2 |   INDEX RANGE SCAN          | ORD_ORDER_DATE_IX  |     1 |       |     2  (50)|
-----------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   2 - access("ORDERS"."ORDER_DATE"=:Z)
```

This should be a highly selective query, and you should see the query using the index on the column to retrieve the desired rows. The data returned is sorted in ascending order by the rowids for the `order_date`. Because the index column `order_date` is identical for the selected rows here, the data is sorted by rowid.

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

- `col1 = :b1`
- `col1 < :b1`
- `col1 > :b1`
- `AND` combination of the preceding conditions for leading columns in the index
- `col1 like 'ASD%'` wild-card searches should not be in a leading position otherwise the condition `col1 like '%ASD'` does not result in a range scan.

Range scans can use unique or non-unique 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`*`)` instructs the optimizer to use a specific index. For more information on the `INDEX` hint, see "Hints for Access Paths" on page 19-3.

Index Range Scans Descending

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

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. For more information on the `INDEX_DESC` hint, see "Hints for Access Paths" on page 19-3.

Index Skip Scans

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

Skip scanning lets a composite index be split logically into smaller subindexes. In skip scanning, the initial column of the composite index is not specified in the query. In other words, it is skipped.

The number of logical subindexes is determined by the number of distinct values in the initial column. Skip scanning is advantageous if there are few distinct values in the leading column of the composite index and many distinct values in the nonleading key of the index.

Example 11–5 Index Skip Scan

Consider, for example, a table `employees` (`sex`, `employee_id`, `address`) with a composite index on (`sex`, `employee_id`). Splitting this composite index would result in two logical subindexes, one for `M` and one for `F`.

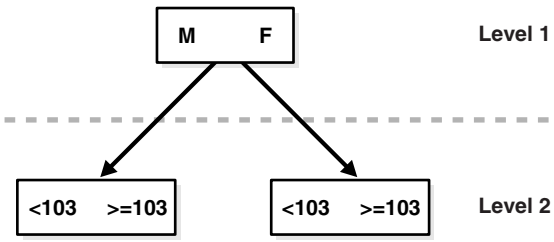
For this example, suppose you have the following index data:

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

The index is split logically into the following two subindexes:

- The first subindex has the keys with the value `F`.
- The second subindex has the keys with the value `M`.

Figure 11–2 Index Skip Scan Illustration



The column `sex` is skipped in the following query:

```
SELECT *
   FROM employees
WHERE employee_id = 101;
```

A complete scan of the index is not performed, but the subindex with the value `F` is searched first, followed by a search of the subindex with the value `M`.

Full Scans

A full scan is available if a predicate references one of the columns in the index. The predicate does not need to be an index driver. A full scan is also available when there is no predicate, if both the following conditions are met:

- All of the columns in the table referenced in the query are included in the index.
- At least one of the index columns is not null.

A full scan 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. A 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.

You can specify fast full index scans 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.

Note: Setting `PARALLEL` for indexes will not impact the cost calculation.

Fast Full Index Scan Hints The fast full scan has a special index hint, `INDEX_FFS`, which has the same format and arguments as the regular `INDEX` hint. For more information on the `INDEX_FFS` hint, see "Hints for Access Paths" on page 19-3.

Index Joins

An index join is a hash join of several indexes that together contain all the table columns 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 Hints You can specify an index join with the `INDEX_JOIN` hint. For more information on the `INDEX_JOIN` hint, see "Hints for Access Paths" on page 19-3.

Bitmap Indexes

A bitmap join 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.

Note: Bitmap indexes and bitmap join indexes are available only if you have purchased the Oracle Enterprise Edition.

See Also: *Oracle Database Data Warehousing Guide* for more information about bitmap indexes

Cluster Access

A cluster scan is used to retrieve, from a table stored in an indexed cluster, all 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 block. 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 Access

A hash scan is used 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 block. 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.

Sample Table Scans

A sample table scan retrieves a random sample of data from a simple table or a complex `SELECT` statement, such as a statement involving joins and views. This access path is used when a statement's `FROM` clause includes the `SAMPLE` clause or the `SAMPLE BLOCK` clause. To perform a sample table scan when sampling by rows with the `SAMPLE` clause, Oracle reads a specified percentage of rows in the table. To perform a sample table scan when sampling by blocks with the `SAMPLE BLOCK` clause, Oracle reads a specified percentage of table blocks.

Example 11–6 uses a sample table scan to access 1% of the `employees` table, sampling by blocks.

Example 11–6 Sample Table Scan

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

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

```
-------------------------------------------------------------------------
| Id  | Operation            |  Name       | Rows  | Bytes | Cost (%CPU)|
-------------------------------------------------------------------------
|   0 | SELECT STATEMENT     |             |     1 |    68 |     3  (34)|
|   1 |  TABLE ACCESS SAMPLE | EMPLOYEES   |     1 |    68 |     3  (34)|
-------------------------------------------------------------------------
```

How the Query Optimizer Chooses an Access Path

The query optimizer 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. 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, the optimizer chooses the execution plan with the lowest estimated cost.

When choosing an access path, the query optimizer is influenced by the following:

- Optimizer Hints

  You can instruct the optimizer to use a specific access path using a hint, except when the statement's FROM clause contains SAMPLE or SAMPLE BLOCK.

  See Also: Chapter 19, "Using Optimizer Hints" for information about hints in SQL statements

- Old Statistics

  For example, if a table has not been analyzed since it was created, and if it has less than DB_FILE_MULTIBLOCK_READ_COUNT blocks under the high water mark, then the optimizer thinks that the table is small and uses a full table scan. Review the LAST_ANALYZED and BLOCKS columns in the ALL_TABLES table to examine the statistics.

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. In a join, one row set is called inner, and the other is called outer.

This section discusses:

- How the Query Optimizer Executes Join Statements
- How the Query Optimizer Chooses Execution Plans for Joins
- Nested Loop Joins
- Hash Joins
- Sort Merge Joins
- Cartesian Joins
- Outer Joins

How the Query Optimizer 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.

- Join Method

  To join each pair of row sources, Oracle must perform a join operation. Join methods include nested loop, sort merge, cartesian, and hash joins.

- 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.

  See Also: "Understanding Access Paths for the Query Optimizer" on page 11-14

How the Query Optimizer Chooses Execution Plans for Joins

The query optimizer considers the following when choosing an execution plan:

- The optimizer first determines whether joining two or more 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. Similarly, when a subquery has been converted into an antijoin or semijoin, the tables from the subquery must come after those tables in the outer query block to which they were connected or correlated. However, hash antijoins and semijoins are able to override this ordering condition in certain circumstances.

With the query optimizer, the optimizer generates a set of execution plans, according to 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 the following 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 cost of a hash join is based largely on the cost of building a hash table on one of the input sides to the join and using the rows from the other of the join to probe it.

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. See "PGA Memory Management" on page 7-42 for information on sizing of SQL work areas.
- A larger multiblock read count is likely to decrease the cost for a sort merge join in relation to a nested loop 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 loop 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 query optimizer, 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 an 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 19, "Using Optimizer Hints" for more information about optimizer hints

Nested Loop Joins

Nested loop joins are useful when small subsets of data are being joined 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 (dependent on) 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, degrading performance considerably. In such cases, hash joins joining the two independent row sources perform better.

See Also: "Cartesian Joins" on page 11-28

A nested loop join involves the following steps:

1. The optimizer determines the driving table and designates it as the outer table.
2. The other table is designated as the inner table.
3. For every row in the outer table, Oracle accesses all the rows in the inner table. The outer loop is for every row in outer table and the inner loop is for every row in the inner table. The outer loop appears before the inner loop in the execution plan, as follows:

```
NESTED LOOPS
  outer_loop
  inner_loop
```

Nested Loop Example

This section discusses the outer and inner loops for one of the nested loops in the query in Example 11–1 on page 11-13.

```
...
|   2 |   NESTED LOOPS                |             |     3 |   141 |     7  (15)|
|*  3 |    TABLE ACCESS FULL          | EMPLOYEES   |     3 |    60 |     4  (25)|
|   4 |    TABLE ACCESS BY INDEX ROWID| JOBS        |    19 |   513 |     2  (50)|
|*  5 |     INDEX UNIQUE SCAN         | JOB_ID_PK   |     1 |       |            |
...
```

In this example, the outer loop retrieves all the rows of the `employees` table. For every employee retrieved by the outer loop, the inner loop retrieves the associated row in the `jobs` table.

Outer loop In the execution plan in Example 11–2 on page 11-13, the outer loop and the equivalent statement are as follows:

```
3 |    TABLE ACCESS FULL        | EMPLOYEES

SELECT e.employee_id, e.salary
  FROM employees e
 WHERE e.employee_id < 103
```

Inner loop The execution plan in Example 11–2 on page 11-13 shows the inner loop being iterated for every row fetched from the outer loop, as follows:

```
4 |    TABLE ACCESS BY INDEX ROWID| JOBS
5 |     INDEX UNIQUE SCAN         | JOB_ID_PK

SELECT j.job_title
 FROM jobs j
    WHERE e.job_id = j.job_id
```

When the Optimizer Uses Nested Loop Joins

The optimizer uses nested loop joins when joining small number of rows, with a good driving condition between the two tables. 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, ideally by 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; for every iteration of the outer loop, the inner loop produces the same set of rows. Therefore, you should use other join methods when two independent row sources are joined together.

Nested Loop Join Hints

If the optimizer is choosing to use some other join method, you can use the `USE_NL(table1 table2)` hint, where `table1` and `table2` are the aliases of the tables being joined.

For some SQL examples, the data is small enough for the optimizer to prefer full table scans and use hash joins. This is the case for the SQL example shown in Example 11–7, "Hash Joins" on page 11-27. However, you can add a `USE_NL` to instruct the optimizer to change the join method to nested loop. For more information on the `USE_NL` hint, see "Hints for Join Operations" on page 19-4.

Nesting Nested Loops

The outer loop of a nested loop can be a nested loop itself. You can nest two or more outer loops together to join as many tables as needed. Each loop is a data access method, as follows:

```
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
```

Hash Joins

Hash joins are used for joining large data sets. The optimizer uses the smaller of two tables or 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 method is best used when the smaller table fits in available memory. The cost is then limited to a single read pass over the data for the two tables.

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 conditions are true:

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

In Example 11–7, the table `orders` is used to build the hash table, and `order_items` is the larger table, which is scanned later.

Example 11–7 Hash Joins

```
SELECT o.customer_id, l.unit_price * l.quantity
  FROM orders o ,order_items l
 WHERE l.order_id = o.order_id;

-------------------------------------------------------------------------------
| Id  | Operation           |  Name        | Rows  | Bytes | Cost (%CPU)|
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT    |              |   665 | 13300 |     8  (25)|
|*  1 |  HASH JOIN          |              |   665 | 13300 |     8  (25)|
|   2 |   TABLE ACCESS FULL | ORDERS       |   105 |   840 |     4  (25)|
|   3 |   TABLE ACCESS FULL | ORDER_ITEMS  |   665 |  7980 |     4  (25)|
-------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   1 - access("L"."ORDER_ID"="O"."ORDER_ID")
```

Hash Join Hints

Apply the `USE_HASH` hint to instruct the optimizer to use a hash join when joining two tables together. See "PGA Memory Management" on page 7-42 for information on sizing of SQL work areas. For more information on the `USE_HASH` hint, see "Hints for Join Operations" on page 19-4.

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. On the other hand, sort merge joins can perform better than hash joins if both of the following conditions exist:

- The row sources are sorted already.
- A sort operation does not have to be done.

However, if a sort merge join involves choosing a slower access method (an index scan as opposed to a 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. The join consists of two steps:

1. Sort join operation: Both the inputs are sorted on the join key.
2. Merge join operation: The sorted lists are merged together.

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

When the Optimizer Uses Sort Merge Joins

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

- The join condition between two tables is not an equi-join.
- Because of sorts already required by other operations, the optimizer finds it is cheaper to use a sort merge than a hash join.

Sort Merge Join Hints

To instruct the optimizer to use a sort merge join, apply the `USE_MERGE` hint. You might also need to give hints to force an access path.

There are situations where it is better to override the optimize with the `USE_MERGE` hint. For example, the optimizer can choose a full scan on a table and avoid a sort operation in a query. However, there is an increased cost because a large table is accessed through an index and single block reads, as opposed to faster access through a full table scan.

For more information on the `USE_MERGE` hint, see "Hints for Join Operations" on page 19-4.

Cartesian Joins

A Cartesian join is used 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, creating the Cartesian product of the two sets.

When the Optimizer Uses Cartesian Joins

The optimizer uses Cartesian joins when it is asked to join two tables with no join conditions. In some cases, a common filter condition between the two tables could be picked up by the optimizer as a possible join condition. In other cases, the optimizer may decide to generate a Cartesian product of two very small tables that are both joined to the same large table.

Cartesian Join Hints

Applying the `ORDERED` hint, instructs the optimizer to use a Cartesian join. By specifying a table before its join table is specified, the optimizer does a Cartesian join.

Outer Joins

An outer join extends the result of a simple join. An outer join returns all rows that satisfy the join condition and also returns some or all of those rows from one table for which no rows from the other satisfy the join condition.

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 a nested loop outer join, the order of tables is determined by the join condition. The outer table, with rows that are being preserved, is used to drive to the inner table.

The optimizer uses nested loop joins to process an outer join in the following circumstances:

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

For an example of a nested loop outer join, you can add the `USE_NL` hint to Example 11–8 to instruct the optimizer to use a nested loop. For example:

```
SELECT /*+ USE_NL(c o) */ cust_last_name, sum(nvl2(o.customer_id,0,1)) "Count"
```

Hash Join Outer Joins

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

The order of tables is determined by the cost. The outer table, including preserved rows, may be used to build the hash table, or it may be used to probe one.

Example 11–8 shows a typical hash join outer join query. In this example, all the customers with credit limits greater than 1000 are queried. An outer join is needed so that you do not miss the customers who do not have any orders.

Example 11–8 Hash Join Outer Joins

```
SELECT cust_last_name, sum(nvl2(o.customer_id,0,1)) "Count"
  FROM customers c, orders o
 WHERE c.credit_limit > 1000
   AND c.customer_id = o.customer_id(+)
 GROUP BY cust_last_name;

-------------------------------------------------------------------------------
| Id  | Operation            |  Name           | Rows  | Bytes | Cost (%CPU)|
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT     |                 |   168 |  3192 |     6  (17)|
|   1 |  HASH GROUP BY       |                 |   168 |  3192 |     6  (17)|
|*  2 |   NESTED LOOPS OUTER |                 |   260 |  4940 |     5  (0) |
|*  3 |    TABLE ACCESS FULL | CUSTOMERS       |   260 |  3900 |     5  (0) |
|*  4 |    INDEX RANGE SCAN  | ORD_CUSTOMER_IX |   105 |   420 |     0  (0) |
-------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   3 - filter("C"."CREDIT_LIMIT">1000)
   4 - access("C"."CUSTOMER_ID"="O"."CUSTOMER_ID"(+))
       filter("O"."CUSTOMER_ID"(+)>0)
```

The query looks for customers which satisfy various conditions. 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 operation finds all the customers rows that do not have any orders rows.

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

```
customers.customer_id = orders.customer_id(+)
```

The components of this condition represent the following:

- The outer table is customers.
- The inner table is orders.
- The join preserves the customers rows, including those rows without a corresponding row in orders.

You could use a NOT EXISTS subquery to return the rows. However, because you are querying all the rows in the table, the hash join performs better (unless the NOT EXISTS subquery is not nested).

In Example 11–9, 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.

Example 11–9 Outer Join to a Multitable View

```
SELECT c.cust_last_name, sum(revenue)
  FROM customers c, v_orders o
 WHERE c.credit_limit > 2000
   AND o.customer_id(+) = c.customer_id
 GROUP BY c.cust_last_name;

-------------------------------------------------------------------------------
| Id  | Operation              |  Name        | Rows  | Bytes | Cost (%CPU)|
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT       |              |   144 |  4608 |    16  (32)|
|   1 |  HASH GROUP BY         |              |   144 |  4608 |    16  (32)|
|*  2 |   HASH JOIN OUTER      |              |   663 | 21216 |    15  (27)|
|*  3 |    TABLE ACCESS FULL   | CUSTOMERS    |   195 |  2925 |     6  (17)|
|   4 |    VIEW                | V_ORDERS     |   665 | 11305 |            |
|   5 |     HASH GROUP BY      |              |   665 | 15960 |     9  (34)|
|*  6 |      HASH JOIN         |              |   665 | 15960 |     8  (25)|
|*  7 |       TABLE ACCESS FULL| ORDERS       |   105 |   840 |     4  (25)|
|   8 |       TABLE ACCESS FULL| ORDER_ITEMS  |   665 | 10640 |     4  (25)|
-------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   2 - access("O"."CUSTOMER_ID"(+)="C"."CUSTOMER_ID")
   3 - filter("C"."CREDIT_LIMIT">2000)
   6 - access("O"."ORDER_ID"="L"."ORDER_ID")
   7 - filter("O"."CUSTOMER_ID">0)
```

The view definition is as follows:

```
CREATE OR REPLACE view v_orders AS
SELECT l.product_id, SUM(l.quantity*unit_price) revenue,
       o.order_id, o.customer_id
  FROM orders o, order_items l
 WHERE o.order_id = l.order_id
 GROUP BY l.product_id, o.order_id, o.customer_id;
```

Sort Merge Outer Joins

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

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.
- The optimizer finds it is cheaper to use a sort merge over a hash join because of sorts already required by other operations.

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 that do not have corresponding rows in the joined tables.

The query in Example 11–10 retrieves all departments and all employees in each department, but also includes:

- Any employees without departments
- Any departments without employees

Example 11–10 Full Outer Join

```
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;
```

The statement produces the following output:

```
DEPARTMENT_ID EMPLOYEE_ID
------------- -----------
           10         200
           20         201
           20         202
           30         114
           30         115
           30         116
...
          270
          280
                     178
                     207

125 rows selected.
```

12

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:

See Also:

- *Oracle Database SQL Reference* for the syntax of the `EXPLAIN PLAN` statement
- Chapter 11, "The Query Optimizer"

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, such as, nested loops join. It also helps you to understand the optimizer decisions, such as why the optimizer chose a nested loops join instead of a hash join, and lets you understand 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 *Oracle Enterprise Manager Concepts*.

How Execution Plans Can Change

With the query optimizer, execution plans can and do change as the underlying optimizer inputs 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.

Note: To avoid possible SQL performance regression that may result from execution plan changes, consider using SQL plan management. For more information, see Chapter 15, "Using SQL Plan Management".

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 and values
- Initialization parameters - set globally or at session level

Minimizing Throw-Away

Examining an explain plan lets you look for throw-away in cases such as the following:

- Full scans
- Unselective range scans
- Late predicate filters
- Wrong join order
- Late filter operations

For example, in the following explain plan, the last step is a very unselective range scan that is executed 76563 times, accesses 11432983 rows, throws away 99% of them, and retains 76563 rows. Why access 11432983 rows to realize that only 76563 rows are needed?

Example 12–1 Looking for Throw-Away in an Explain Plan

```
Rows      Execution Plan
--------  -------------------------------------------------
      12  SORT AGGREGATE
       2   SORT GROUP BY
   76563    NESTED LOOPS
   76575     NESTED LOOPS
      19      TABLE ACCESS FULL CN_PAYRUNS_ALL
   76570      TABLE ACCESS BY INDEX ROWID CN_POSTING_DETAILS_ALL
   76570       INDEX RANGE SCAN (object id 178321)
   76563     TABLE ACCESS BY INDEX ROWID CN_PAYMENT_WORKSHEETS_ALL
11432983      INDEX RANGE SCAN (object id 186024)
```

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.

Using V$SQL_PLAN Views

In addition to running the `EXPLAIN PLAN` command and displaying the plan, you can use the `V$SQL_PLAN` views to display the execution plan of a SQL statement:

After the statement has executed, you can display the plan by querying the `V$SQL_PLAN` view. `V$SQL_PLAN` contains the execution plan for every statement stored in the cursor cache. Its definition is similar to the `PLAN_TABLE`. See "PLAN_TABLE Columns" on page 12-18.

The advantage of `V$SQL_PLAN` over `EXPLAIN PLAN` is that you do not need to know the compilation environment that was used to execute a particular statement. For

EXPLAIN PLAN, you would need to set up an identical environment to get the same plan when executing the statement.

The V$SQL_PLAN_STATISTICS view provides the actual execution statistics for every operation in the plan, such as the number of output rows and elapsed time. All statistics, except the number of output rows, are cumulative. For example, the statistics for a join operation also includes the statistics for its two inputs. The statistics in V$SQL_PLAN_STATISTICS are available for cursors that have been compiled with the STATISTICS_LEVEL initialization parameter set to ALL.

The V$SQL_PLAN_STATISTICS_ALL view enables side by side comparisons of the estimates that the optimizer provides for the number of rows and elapsed time. This view combines both V$SQL_PLAN and V$SQL_PLAN_STATISTICS information for every cursor.

See Also:

- "Real-Time SQL Monitoring" on page 10-37 for information about the V$SQL_PLAN_MONITOR view

  Oracle Database Reference for more information about V$SQL_PLAN views
- *Oracle Database Reference* for information about the STATISTICS_LEVEL initialization parameter

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 21, "Using Application Tracing Tools"

The PLAN_TABLE Output Table

The PLAN_TABLE is automatically created as a global temporary table to hold the output of an EXPLAIN PLAN statement for all users. PLAN_TABLE is the default sample output table into which the EXPLAIN PLAN statement inserts rows describing execution plans. See "PLAN_TABLE Columns" on page 12-18 for a description of the columns in the table.

While a PLAN_TABLE table is automatically set up for each user, you can use the SQL script utlxplan.sql to manually create a local PLAN_TABLE in your schema. The exact name and location of this script depends on your operating system. On Unix, it is located in the $ORACLE_HOME/rdbms/admin directory.

For example, run the commands in Example 12–2 from a SQL*Plus session to create the PLAN_TABLE in the HR schema.

Example 12–2 Creating a PLAN_TABLE

```
CONNECT HR/your_password
@$ORACLE_HOME/rdbms/admin/utlxplan.sql

Table created.
```

Oracle recommends that you drop and rebuild your local PLAN_TABLE table after upgrading the version of the database because the columns might change. This can cause scripts to fail or cause TKPROF to fail, if you are specifying the table.

If you want an output table with a different name, first create PLAN_TABLE manually with the utlxplan.sql script and then rename the table with the RENAME SQL statement. For example:

```
RENAME PLAN_TABLE TO my_plan_table;
```

Running EXPLAIN PLAN

To explain a SQL statement, use the EXPLAIN PLAN FOR clause immediately before the statement. For example:

```
EXPLAIN PLAN FOR
  SELECT last_name FROM employees;
```

This explains the plan into the PLAN_TABLE table. You can then select the execution plan from PLAN_TABLE. See "Displaying PLAN_TABLE Output" on page 12-6.

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 SET STATEMENT ID, remove any existing rows for that statement ID.

In Example 12–3, st1 is specified as the statement identifier:

Example 12–3 Using EXPLAIN PLAN with the STATEMENT ID Clause

```
EXPLAIN PLAN
  SET STATEMENT_ID = 'st1' FOR
SELECT last_name FROM employees;
```

Specifying Different Tables for EXPLAIN PLAN

You can specify the INTO clause to specify a different table.

Example 12–4 Using EXPLAIN PLAN with the INTO Clause

```
EXPLAIN PLAN
  INTO my_plan_table
 FOR
SELECT last_name FROM employees;
```

You can specify a statement Id when using the INTO clause.

```
EXPLAIN PLAN
   SET STATEMENT_ID = 'st1'
   INTO my_plan_table
 FOR
SELECT last_name FROM employees;
```

See Also: *Oracle Database SQL Reference* for a complete description of EXPLAIN PLAN syntax.

Displaying PLAN_TABLE Output

After you have explained the plan, use the following SQL scripts or PL/SQL package provided by Oracle to display the most recent plan table output:

- UTLXPLS.SQL

  This script displays the plan table output for serial processing. Example 11–2, "EXPLAIN PLAN Output" on page 11-13 is an example of the plan table output when using the UTLXPLS.SQL script.

- UTLXPLP.SQL

  This script displays the plan table output including parallel execution columns.

- DBMS_XPLAN.DISPLAY procedure

  This procedure accepts options for displaying the plan table output. You can specify:

  - A plan table name if you are using a table different than PLAN_TABLE
  - A statement Id if you have set a statement Id with the EXPLAIN PLAN
  - A format option that determines the level of detail: BASIC, SERIAL, and TYPICAL, ALL,

  Some examples of the use of DBMS_XPLAN to display PLAN_TABLE output are:

```
SELECT PLAN_TABLE_OUTPUT FROM TABLE(DBMS_XPLAN.DISPLAY());

SELECT PLAN_TABLE_OUTPUT
  FROM TABLE(DBMS_XPLAN.DISPLAY('MY_PLAN_TABLE', 'st1','TYPICAL'));
```

See Also: *Oracle Database PL/SQL Packages and Types Reference* for more information on the DBMS_XPLAN package

Customizing PLAN_TABLE Output

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 = 'st1'
  ORDER BY id;

   Rows Plan
------- ---------------------------------------
        SELECT STATEMENT
         TABLE ACCESS FULL EMPLOYEES
```

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 EMPLOYEES
```

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.

Note: These simplified examples are not valid for recursive SQL.

Reading EXPLAIN PLAN Output

This section uses EXPLAIN PLAN examples to illustrate execution plans. The statement in Example 12–5 is used to display the execution plans.

Example 12–5 Statement to display the EXPLAIN PLAN

```
SELECT PLAN_TABLE_OUTPUT
  FROM TABLE(DBMS_XPLAN.DISPLAY(NULL, 'statement_id','BASIC'));
```

Examples of the output from this statement are shown in Example 12–6 and Example 12–7.

Example 12–6 EXPLAIN PLAN for Statement Id ex_plan1

```
EXPLAIN PLAN
  SET statement_id = 'ex_plan1' FOR
SELECT phone_number FROM employees
 WHERE phone_number LIKE '650%';

---------------------------------------
| Id  | Operation         | Name      |
---------------------------------------
|   0 | SELECT STATEMENT  |           |
|   1 |  TABLE ACCESS FULL| EMPLOYEES |
---------------------------------------
```

This plan shows execution of a SELECT statement. The table employees is accessed using a full table scan.

- Every row in the table employees is accessed, and the WHERE clause criteria is evaluated for every row.
- The SELECT statement returns the rows meeting the WHERE clause criteria.

Example 12–7 EXPLAIN PLAN for Statement Id ex_plan2

```
EXPLAIN PLAN
  SET statement_id = 'ex_plan2' FOR
SELECT last_name FROM employees
 WHERE last_name LIKE 'Pe%';

SELECT PLAN_TABLE_OUTPUT
  FROM TABLE(DBMS_XPLAN.DISPLAY(NULL, 'ex_plan2','BASIC'));

---------------------------------------
```

```
| Id  | Operation         | Name        |
---------------------------------------
|   0 | SELECT STATEMENT  |             |
|   1 |  INDEX RANGE SCAN | EMP_NAME_IX |
---------------------------------------
```

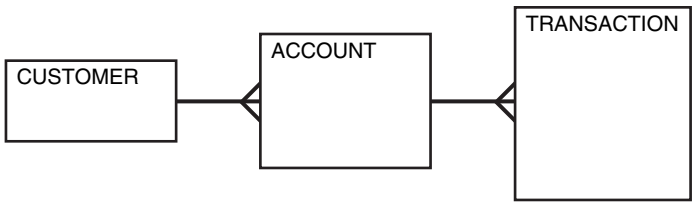
This plan shows execution of a SELECT statement.

- Index EMP_NAME_IX is used in a range scan operation to evaluate the WHERE clause criteria.
- The SELECT statement returns rows satisfying the WHERE clause conditions.

Viewing Parallel Execution with EXPLAIN PLAN

Tuning a parallel query begins much like a non-parallel query tuning exercise by choosing the driving table. However, the rules governing the choice are different. In the non-parallel case, the best driving table is typically the one that produces fewest number of rows after limiting conditions are applied. The small number of rows are joined to larger tables using non-unique indexes. For example, consider a table hierarchy consisting of CUSTOMER, ACCOUNT, and TRANSACTION.

Figure 12–1 A Table Hierarchy



CUSTOMER is the smallest table while TRANSACTION is the largest. A typical OLTP query might be to retrieve transaction information about a particular customer's account. The query would drive from the CUSTOMER table. The goal in this case is to minimize logical I/O, which typically minimizes other critical resources including physical I/O and CPU time.

For parallel queries, the choice of the driving table is usually the largest table because parallel query can be utilized. Obviously, it would not be efficient to use parallel query on the query, because only a few rows from each table are ultimately accessed. However, what if it were necessary to identify all customers that had transactions of a certain type last month? It would be more efficient to drive from the TRANSACTION table because there are no limiting conditions on the customer table. The rows from the TRANSACTION table would be joined to the ACCOUNT table, and finally to the CUSTOMER table. In this case, the indexes utilized on the ACCOUNT and CUSTOMER table are likely to be highly selective primary key or unique indexes, rather than non-unique indexes used in the first query. Because the TRANSACTION table is large and the column is un-selective, it would be beneficial to utilize parallel query driving from the TRANSACTION table.

Parallel operations include:

- PARALLEL_TO_PARALLEL
- PARALLEL_TO_SERIAL

  A PARALLEL_TO_SERIAL operation which is always the step that occurs when rows from a parallel operation are consumed by the query coordinator. Another

type of operation that does not occur in this query is a SERIAL operation. If these types of operations occur, consider making them parallel operations to improve performance because they too are potential bottlenecks.

- PARALLEL_FROM_SERIAL
- PARALLEL_TO_PARALLEL

  PARALLEL_TO_PARALLEL operations generally produce the best performance as long as the workloads in each step are relatively equivalent.

- PARALLEL_COMBINED_WITH_CHILD
- PARALLEL_COMBINED_WITH_PARENT

  A PARALLEL_COMBINED_WITH_PARENT operation occurs when the step is performed simultaneously with the parent step.

If a parallel step produces many rows, the query coordinator (QC) may not be able to consume them as fast as they are being produced. There is little that can be done to improve this.

> **See Also:** See the OTHER_TAG column in Table 12–1, " PLAN_ TABLE Columns" on page 12-18

Viewing Parallel Queries with EXPLAIN PLAN

When using EXPLAIN PLAN with parallel queries, one parallel plan is compiled and executed. This plan is derived from the serial plan by allocating row sources specific to the parallel support in the Query Coordinator (QC) plan. The table queue row sources (PX Send and PX Receive), the granule iterator, and buffer sorts, required by the two slave set PQ model, are directly inserted into the parallel plan. This plan is the exact same plan for all the slaves if executed in parallel or for the QC if executed in serial.

Example 12–8 is a simple query for illustrating an EXPLAIN PLAN for a parallel query.

Example 12–8 Parallel Query Explain Plan

```
CREATE TABLE emp2 AS SELECT * FROM employees;
ALTER TABLE emp2 PARALLEL 2;

EXPLAIN PLAN FOR
  SELECT SUM(salary) FROM emp2 GROUP BY department_id;
SELECT PLAN_TABLE_OUTPUT FROM TABLE(DBMS_XPLAN.DISPLAY());
```

```
-------------------------------------------------------------------------------------------------------
| Id  | Operation               | Name     | Rows  | Bytes | Cost (%CPU) |    TQ  |IN-OUT| PQ Distrib |
-------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT        |          |   107 |  2782 |     3 (34)  |        |      |            |
|   1 |  PX COORDINATOR         |          |       |       |             |        |      |            |
|   2 |   PX SEND QC (RANDOM)   | :TQ10001 |   107 |  2782 |     3 (34)  |  Q1,01 | P->S | QC (RAND)  |
|   3 |    HASH GROUP BY        |          |   107 |  2782 |     3 (34)  |  Q1,01 | PCWP |            |
|   4 |     PX RECEIVE          |          |   107 |  2782 |     3 (34)  |  Q1,01 | PCWP |            |
|   5 |      PX SEND HASH       | :TQ10000 |   107 |  2782 |     3 (34)  |  Q1,00 | P->P | HASH       |
|   6 |       HASH GROUP BY     |          |   107 |  2782 |     3 (34)  |  Q1,00 | PCWP |            |
|   7 |        PX BLOCK ITERATOR|          |   107 |  2782 |     2 (0)   |  Q1,00 | PCWP |            |
|   8 |         TABLE ACCESS FULL| EMP2     |   107 |  2782 |     2 (0)   |  Q1,00 | PCWP |            |
-------------------------------------------------------------------------------------------------------
```

The table EMP2 is scanned in parallel by one set of slaves while the aggregation for the GROUP BY is done by the second set. The PX BLOCK ITERATOR row source represents the splitting up of the table EMP2 into pieces so as to divide the scan workload between the parallel scan slaves. The PX SEND and PX RECEIVE row sources represent

the pipe that connects the two slave sets as rows flow up from the parallel scan, get repartitioned through the HASH table queue, and then read by and aggregated on the top slave set. The PX SEND QC row source represents the aggregated values being sent to the QC (Query Coordinator) in random (RAND) order. The PX COORDINATOR row source represents the QC or Query Coordinator which controls and schedules the parallel plan appearing below it in the plan tree.

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 sample query and plan in Example 12–9.

Example 12–9 EXPLAIN PLAN with Bitmap Indexes

```
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 Result Cache with EXPLAIN PLAN

When your query contains the result_cache hint, the ResultCache operator is inserted into the execution plan.

For example, consider the query:

```
select /*+ result_cache */ deptno, avg(sal)
from emp
group by deptno;
```

To view the EXPLAIN PLAN for this query, use the command:

```
EXPLAIN PLAN FOR
```

```
select /*+ result_cache */ deptno, avg(sal)
from emp
group by deptno;

select PLAN_TABLE_OUTPUT from TABLE (DBMS_XPLAN.DISPLAY());
```

The EXPLAIN PLAN output for this query will be similar to:

```
-------------------------------------------------------------------------------------------
| Id  | Operation           | Name                         |Rows |Bytes |Cost(%CPU)|Time     |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT    |                              | 11  | 77   |  4 (25)  | 00:00:01|
|   1 |  RESULT CACHE       |b06ppfz9pxzstbttpbqyqnfbmy    |     |      |          |         |
|   2 |   HASH GROUP BY     |                              | 11  | 77   |  4 (25)  | 00:00:01|
|   3 |    TABLE ACCESS FULL| EMP                          |107  | 749  |  3 (0)   | 00:00:01|
-------------------------------------------------------------------------------------------
```

In this EXPLAIN PLAN, the ResultCache operator is identified by its CacheId, which is b06ppfz9pxzstbttpbqyqnfbmy. You can now run a query on the V$RESULT_CACHE_OBJECTS view by using this CacheId.

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 hire_date to illustrate how pruning is displayed. Assume that the tables employees and departments from the Oracle sample schema exist.

```
CREATE TABLE emp_range
PARTITION BY RANGE(hire_date)
(
PARTITION emp_p1 VALUES LESS THAN (TO_DATE('1-JAN-1992','DD-MON-YYYY')),
PARTITION emp_p2 VALUES LESS THAN (TO_DATE('1-JAN-1994','DD-MON-YYYY')),
PARTITION emp_p3 VALUES LESS THAN (TO_DATE('1-JAN-1996','DD-MON-YYYY')),
PARTITION emp_p4 VALUES LESS THAN (TO_DATE('1-JAN-1998','DD-MON-YYYY')),
PARTITION emp_p5 VALUES LESS THAN (TO_DATE('1-JAN-2001','DD-MON-YYYY'))
)
AS SELECT * FROM employees;
```

For the first example, consider the following statement:

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

Oracle displays something similar to the following:

```
-------------------------------------------------------------------------------
| Id  | Operation           | Name      | Rows  | Bytes | Cost  | Pstart| Pstop |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT    |           |   105 | 13965 |     2 |       |       |
|   1 |  PARTITION RANGE ALL|           |   105 | 13965 |     2 |     1 |     5 |
|   2 |   TABLE ACCESS FULL | EMP_RANGE |   105 | 13965 |     2 |     1 |     5 |
-------------------------------------------------------------------------------
```

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 this 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 hire_date >= TO_DATE('1-JAN-1996','DD-MON-YYYY');

-------------------------------------------------------------------------------
| Id  | Operation               | Name      | Rows  | Bytes | Cost  | Pstart| Pstop |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT        |           |     3 |   399 |     2 |       |       |
|   1 |  PARTITION RANGE ITERATOR|           |     3 |   399 |     2 |     4 |     5 |
|*  2 |   TABLE ACCESS FULL     | EMP_RANGE |     3 |   399 |     2 |     4 |     5 |
-------------------------------------------------------------------------------
```

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

Finally, consider the following statement:

```
EXPLAIN PLAN FOR
  SELECT * FROM emp_range
  WHERE hire_date < TO_DATE('1-JAN-1992','DD-MON-YYYY');

-------------------------------------------------------------------------------
| Id  | Operation             | Name      | Rows  | Bytes | Cost  | Pstart| Pstop |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT      |           |     1 |   133 |     2 |       |       |
|   1 |  PARTITION RANGE SINGLE|           |     1 |   133 |     2 |     1 |     1 |
|*  2 |   TABLE ACCESS FULL   | EMP_RANGE |     1 |   133 |     2 |     1 |     1 |
-------------------------------------------------------------------------------
```

In the previous 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(hire_date)
      SUBPARTITION BY HASH(department_id) SUBPARTITIONS 3
```

```
(
PARTITION emp_p1 VALUES LESS THAN (TO_DATE('1-JAN-1992','DD-MON-YYYY')),
PARTITION emp_p2 VALUES LESS THAN (TO_DATE('1-JAN-1994','DD-MON-YYYY')),
PARTITION emp_p3 VALUES LESS THAN (TO_DATE('1-JAN-1996','DD-MON-YYYY')),
PARTITION emp_p4 VALUES LESS THAN (TO_DATE('1-JAN-1998','DD-MON-YYYY')),
PARTITION emp_p5 VALUES LESS THAN (TO_DATE('1-JAN-2001','DD-MON-YYYY'))
)
AS SELECT * FROM employees;
```

For the first example, consider the following statement:

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

```
-------------------------------------------------------------------------------
| Id  | Operation           | Name     | Rows  | Bytes | Cost  | Pstart| Pstop |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT    |          | 10120 |  1314K|    78 |       |       |
|   1 |  PARTITION RANGE ALL|          | 10120 |  1314K|    78 |     1 |     5 |
|   2 |   PARTITION HASH ALL|          | 10120 |  1314K|    78 |     1 |     3 |
|   3 |    TABLE ACCESS FULL| EMP_COMP | 10120 |  1314K|    78 |     1 |    15 |
-------------------------------------------------------------------------------
```

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 hire_date = TO_DATE('15-FEB-1998', 'DD-MON-YYYY');
```

```
-------------------------------------------------------------------------------
| Id  | Operation             | Name     | Rows  | Bytes | Cost  | Pstart| Pstop |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT      |          |    20 |  2660 |    17 |       |       |
|   1 |  PARTITION RANGE SINGLE|          |    20 |  2660 |    17 |     5 |     5 |
|   2 |   PARTITION HASH ALL  |          |    20 |  2660 |    17 |     1 |     3 |
|*  3 |    TABLE ACCESS FULL  | EMP_COMP |    20 |  2660 |    17 |    13 |    15 |
-------------------------------------------------------------------------------
```

In the previous 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 department_id = 20;
```

```
-------------------------------------------------------------------------------
| Id  | Operation             | Name     | Rows  | Bytes | Cost  | Pstart| Pstop |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT      |          |   101 | 13433 |    78 |       |       |
|   1 |  PARTITION RANGE ALL  |          |   101 | 13433 |    78 |     1 |     5 |
|   2 |   PARTITION HASH SINGLE|          |   101 | 13433 |    78 |     3 |     3 |
```

```
|*  3 |    TABLE ACCESS FULL   | EMP_COMP |   101 | 13433 |    78 |       |       |
-------------------------------------------------------------------------------
```

In the previous 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 department_id = :dno;

-------------------------------------------------------------------------------
| Id  | Operation              | Name     | Rows  | Bytes | Cost  | Pstart| Pstop |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT       |          |   101 | 13433 |    78 |       |       |
|   1 |  PARTITION RANGE ALL   |          |   101 | 13433 |    78 |     1 |     5 |
|   2 |   PARTITION HASH SINGLE|          |   101 | 13433 |    78 |   KEY |   KEY |
|*  3 |    TABLE ACCESS FULL   | EMP_COMP |   101 | 13433 |    78 |       |       |
-------------------------------------------------------------------------------
```

The last two examples are the same, except that `deptno = 20` has been replaced by `department_id = :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_did` is joined on the partitioning column `department_id` and is parallelized. This enables use of partial partition-wise join, because the `dept2` table is not partitioned. Oracle dynamically partitions the dept2 table before the join.

Example 12–10 Partial Partition-Wise Join with Range Partition

```
CREATE TABLE dept2 AS SELECT * FROM departments;
ALTER TABLE dept2 PARALLEL 2;

CREATE TABLE emp_range_did PARTITION BY RANGE(department_id)
   (PARTITION emp_p1 VALUES LESS THAN (150),
    PARTITION emp_p5 VALUES LESS THAN (MAXVALUE) )
  AS SELECT * FROM employees;

ALTER TABLE emp_range_did PARALLEL 2;

EXPLAIN PLAN FOR
SELECT /*+ PQ_DISTRIBUTE(d NONE PARTITION) ORDERED */ e.last_name,
                     d.department_name
   FROM emp_range_did e , dept2 d
   WHERE e.department_id = d.department_id ;

-------------------------------------------------------------------------------------------------------
| Id| Operation                  |Name         |Rows | Bytes |Cost|Pstart|Pstop|   TQ  |IN-OUT|PQ Distrib |
-------------------------------------------------------------------------------------------------------
|  0| SELECT STATEMENT           |             | 284 | 16188 |  6 |      |     |       |      |           |
|  1|  PX COORDINATOR            |             |     |       |    |      |     |       |      |           |
|  2|   PX SEND QC (RANDOM)      |:TQ10001     | 284 | 16188 |  6 |      |     | Q1,01 | P->S | QC (RAND) |
```

```
|* 3|    HASH JOIN                 |             | 284 | 16188 |  6 |      |     | Q1,01 | PCWP |           |
|  4|     PX PARTITION RANGE ALL   |             | 284 |  7668 |  2 |    1 |   2 | Q1,01 | PCWC |           |
|  5|      TABLE ACCESS FULL       |EMP_RANGE_DID| 284 |  7668 |  2 |    1 |   2 | Q1,01 | PCWP |           |
|  6|     BUFFER SORT              |             |     |       |    |      |     | Q1,01 | PCWC |           |
|  7|      PX RECEIVE              |             |  21 |   630 |  2 |      |     | Q1,01 | PCWP |           |
|  8|       PX SEND PARTITION (KEY)|:TQ10000     |  21 |   630 |  2 |      |     |       | S->P |PART (KEY) |
|  9|        TABLE ACCESS FULL     |DEPT2        |  21 |   630 |  2 |      |     |       |      |           |
-------------------------------------------------------------------------------------------------------------
```

The execution plan shows that the table `dept2` is scanned serially and all rows with the same partitioning column value of `emp_range_did` (`department_id`) are sent through a `PART` (`KEY`), or partition key, table queue to the same slave doing the partial partition-wise join.

In the following example, `emp_comp` is joined on the partitioning column and is parallelized. This enables use of partial partition-wise join, because the `dept2` table is not partitioned. Oracle dynamically partitions the `dept2` table before the join.

Example 12–11 Partial Partition-Wise Join with Composite Partition

```
ALTER TABLE emp_comp PARALLEL 2;

EXPLAIN PLAN FOR
SELECT /*+ PQ_DISTRIBUTE(d NONE PARTITION) ORDERED */ e.last_name,
          d.department_name
  FROM emp_comp e, dept2 d
 WHERE e.department_id = d.department_id;
SELECT PLAN_TABLE_OUTPUT FROM TABLE(DBMS_XPLAN.DISPLAY());
```

```
-----------------------------------------------------------------------------------------------------------
| Id | Operation                  | Name     | Rows | Bytes | Cost |Pstart|Pstop|    TQ  |IN-OUT| PQ Distrib |
-----------------------------------------------------------------------------------------------------------
|  0 | SELECT STATEMENT           |          |  445 | 17800 |    5 |      |     |        |      |            |
|  1 |  PX COORDINATOR            |          |      |       |      |      |     |        |      |            |
|  2 |   PX SEND QC (RANDOM)      |:TQ10001  |  445 | 17800 |    5 |      |     |  Q1,01 | P->S | QC (RAND)  |
|* 3 |    HASH JOIN               |          |  445 | 17800 |    5 |      |     |  Q1,01 | PCWP |            |
|  4 |     PX PARTITION RANGE ALL |          |  107 |  1070 |    3 |    1 |   5 |  Q1,01 | PCWC |            |
|  5 |      PX PARTITION HASH ALL |          |  107 |  1070 |    3 |    1 |   3 |  Q1,01 | PCWC |            |
|  6 |       TABLE ACCESS FULL    |EMP_COMP  |  107 |  1070 |    3 |    1 |  15 |  Q1,01 | PCWP |            |
|  7 |     PX RECEIVE             |          |   21 |   630 |    1 |      |     |  Q1,01 | PCWP |            |
|  8 |      PX SEND PARTITION (KEY)|:TQ10000 |   21 |   630 |    1 |      |     |  Q1,00 | P->P |PART (KEY)  |
|  9 |       PX BLOCK ITERATOR    |          |   21 |   630 |    1 |      |     |  Q1,00 | PCWC |            |
| 10 |        TABLE ACCESS FULL   |DEPT2     |   21 |   630 |    1 |      |     |  Q1,00 | PCWP |            |
-----------------------------------------------------------------------------------------------------------
```

The plan shows that the optimizer selects partial partition-wise join from one of two columns. The `PX SEND` node type is `PARTITION(KEY)` and the PQ Distrib column contains the text `PART (KEY)`, or partition key. This implies that the table `dept2` is re-partitioned based on the join column `department_id` to be sent to the parallel slaves executing the scan of `EMP_COMP` and the join.

Note that in both Example 12–10 and Example 12–11 the `PQ_DISTRIBUTE` hint is used to explicitly force a partial partition-wise join because the query optimizer could have chosen a different plan based on cost in this query.

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.

The PX PARTITION HASH row source appears on top of the join row source in the plan table output while the PX PARTITION RANGE row source appears over the scan of emp_comp. Each parallel slave performs the join of an entire hash partition of emp_comp with an entire partition of dept_hash.

Example 12–12 Full Partition-Wise Join

```
CREATE TABLE dept_hash
   PARTITION BY HASH(department_id)
   PARTITIONS 3
   PARALLEL 2
   AS SELECT * FROM departments;

EXPLAIN PLAN FOR SELECT /*+ PQ_DISTRIBUTE(e NONE NONE) ORDERED */ e.last_name,
       d.department_name
  FROM emp_comp e, dept_hash d
 WHERE e.department_id = d.department_id;
```

```
-------------------------------------------------------------------------------------------------------
| Id | Operation                 | Name      | Rows |Bytes |Cost |Pstart|Pstop |   TQ  |IN-OUT| PQ Distrib |
-------------------------------------------------------------------------------------------------------
|  0 | SELECT STATEMENT          |           |  106 | 2544 |   8 |      |      |       |      |            |
|  1 |  PX COORDINATOR           |           |      |      |     |      |      |       |      |            |
|  2 |   PX SEND QC (RANDOM)     | :TQ10000  |  106 | 2544 |   8 |      |      | Q1,00 | P->S | QC (RAND)  |
|  3 |    PX PARTITION HASH ALL  |           |  106 | 2544 |   8 |    1 |    3 | Q1,00 | PCWC |            |
|* 4 |     HASH JOIN             |           |  106 | 2544 |   8 |      |      | Q1,00 | PCWP |            |
|  5 |      PX PARTITION RANGE ALL|          |  107 | 1070 |   3 |    1 |    5 | Q1,00 | PCWC |            |
|  6 |       TABLE ACCESS FULL   | EMP_COMP  |  107 | 1070 |   3 |    1 |   15 | Q1,00 | PCWP |            |
|  7 |      TABLE ACCESS FULL    | DEPT_HASH |   27 |  378 |   4 |    1 |    3 | Q1,00 | PCWP |            |
-------------------------------------------------------------------------------------------------------
```

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 next operation in the plan 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 before the table operation but after 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 before 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
```

PLAN_TABLE Columns

The PLAN_TABLE used by the EXPLAIN PLAN statement contains the columns listed in Table 12–1.

Table 12–1 PLAN_TABLE Columns

Column	Type	Description
STATEMENT_ID	VARCHAR2(30)	Value of the optional STATEMENT_ID parameter specified in the EXPLAIN PLAN statement.
PLAN_ID	NUMBER	Unique identifier of a plan in the database.
TIMESTAMP	DATE	Date and time when the EXPLAIN PLAN statement was generated.
REMARKS	VARCHAR2(80)	Any comment (of up to 80 bytes) you want to associate with each step of the explained plan. This column is used to indicate whether an outline or SQL Profile was used for the query. 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)	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 12–3 for more information on values for this column.
OPTIONS	VARCHAR2(225)	A variation on the operation described in the OPERATION column. See Table 12–3 for more information on values for this column.
OBJECT_NODE	VARCHAR2(128)	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)	Name of the user who owns the schema containing the table or index.
OBJECT_NAME	VARCHAR2(30)	Name of the table or index.
OBJECT_ALIAS	VARCHAR2(65)	Unique alias of a table or view in a SQL statement. For indexes, it is the object alias of the underlying table.
OBJECT_INSTANCE	NUMERIC	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)	Modifier that provides descriptive information about the object; for example, NON-UNIQUE for indexes.
OPTIMIZER	VARCHAR2(255)	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.

Table 12–1 (Cont.) PLAN_TABLE Columns

Column	Type	Description
DEPTH	NUMERIC	Depth of the operation in the row source tree that the plan represents. The value can be used for indenting the rows in a plan table report.
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	Cost of the operation as estimated by the optimizer's query approach. 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	Estimate by the query optimization approach of the number of rows accessed by the operation.
BYTES	NUMERIC	Estimate by the query optimization approach of the number of bytes accessed by the operation.
OTHER_TAG	VARCHAR2(255)	Describes the contents of the OTHER column. Values are: ■ SERIAL (blank) - Serial execution. Currently, SQL is not loaded in the OTHER column for this case. ■ SERIAL_FROM_REMOTE (S -> R) - Serial execution at a remote site. ■ PARALLEL_FROM_SERIAL (S -> P) - Serial execution. Output of step is partitioned or broadcast to parallel execution servers. ■ PARALLEL_TO_SERIAL (P -> S) - Parallel execution. Output of step is returned to serial query coordinator (QC) process. ■ PARALLEL_TO_PARALLEL (P -> P) - Parallel execution. Output of step is repartitioned to second set of parallel execution servers. ■ PARALLEL_COMBINED_WITH_PARENT (PWP) - Parallel execution; Output of step goes to next step in same parallel process. No interprocess communication to parent. ■ PARALLEL_COMBINED_WITH_CHILD (PWC) - Parallel execution. Input of step comes from prior step in same parallel process. No interprocess communication from child.
PARTITION_START	VARCHAR2(255)	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 REMOVE_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.

Table 12–1 (Cont.) PLAN_TABLE Columns

Column	Type	Description
PARTITION_STOP	VARCHAR2(255)	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 REMOVE_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	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. See the OTHER_TAG column.
DISTRIBUTION	VARCHAR2(30)	Method used to distribute rows from producer query servers to consumer query servers. See Table 12–2 for more information on the possible values for this column. For more information about consumer and producer query servers, see *Oracle Database Data Warehousing Guide*.
CPU_COST	NUMERIC	CPU cost of the operation as estimated by the query optimizer's approach. The value of this column is proportional to the number of machine cycles required for the operation. For statements that use the rule-based approach, this column is null.
IO_COST	NUMERIC	I/O cost of the operation as estimated by the query optimizer's approach. The value of this column is proportional to the number of data blocks read by the operation. For statements that use the rule-based approach, this column is null.
TEMP_SPACE	NUMERIC	Temporary space, in bytes, used by the operation as estimated by the query optimizer's approach. For statements that use the rule-based approach, or for operations that do not use any temporary space, this column is null.
ACCESS_PREDICATES	VARCHAR2(4000)	Predicates used to locate rows in an access structure. For example, start or stop predicates for an index range scan.
FILTER_PREDICATES	VARCHAR2(4000)	Predicates used to filter rows before producing them.
PROJECTION	VARCHAR2(4000)	Expressions produced by the operation.
TIME	NUMBER(20,2)	Elapsed time in seconds of the operation as estimated by query optimization. For statements that use the rule-based approach, this column is null.
QBLOCK_NAME	VARCHAR2(30)	Name of the query block, either system-generated or defined by the user with the QB_NAME hint.

Table 12–2 describes the values that can appear in the DISTRIBUTION column:

Table 12–2 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 (QC) 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 (QC) consumes the input randomly. Used when the statement does not have an ORDER BY clause.

Table 12–3 lists each combination of OPERATION and OPTIONS produced by the EXPLAIN PLAN statement and its meaning within an execution plan.

Table 12–3 OPERATION and OPTIONS Values Produced by EXPLAIN PLAN

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.
BITMAP	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.
BITMAP	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.
BITMAP	MERGE	Merges several bitmaps resulting from a range scan into one bitmap.
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 12-10.
BITMAP	OR	Computes the bitwise OR of two bitmaps.
BITMAP	AND	Computes the bitwise AND of two bitmaps.

Table 12–3 (Cont.) OPERATION and OPTIONS Values Produced by EXPLAIN PLAN

Operation	Option	Description
BITMAP	KEY ITERATION	Takes each row from a table row source and finds the corresponding bitmap from a bitmap index. This set of bitmaps are then merged into one bitmap in a following BITMAP MERGE operation.
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.
COUNT	STOPKEY	Count operation where the number of rows returned is limited by the ROWNUM expression in the WHERE clause.
CUBE SCAN	.	Uses inner joins for all cube access.
CUBE SCAN	PARTIAL OUTER	Uses an outer join for at least one dimension, and inner joins for the other dimensions.
CUBE SCAN	OUTER	Uses outer joins for all cube access.
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.
FOR UPDATE	.	Operation retrieving and locking the rows selected by a query containing a FOR UPDATE clause.
HASH	GROUP BY	Operation hashing a set of rows into groups for a query with a GROUP BY clause.
HASH JOIN (These are join operations.)	.	Operation joining two sets of rows and returning the result. This join method is useful for joining large data sets of data (DSS, Batch). The join condition is an efficient way of accessing the second table. Query optimizer uses the smaller of the two tables/data sources to build a hash table on the join key in memory. Then it scans the larger table, probing the hash table to find the joined rows.
HASH JOIN	ANTI	Hash (left) antijoin
HASH JOIN	SEMI	Hash (left) semijoin
HASH JOIN	RIGHT ANTI	Hash right antijoin
HASH JOIN	RIGHT SEMI	Hash right semijoin
HASH JOIN	OUTER	Hash (left) outer join
HASH JOIN	RIGHT OUTER	Hash right outer join
INDEX (These are access methods.)	UNIQUE SCAN	Retrieval of a single rowid from an index.
INDEX	RANGE SCAN	Retrieval of one or more rowids from an index. Indexed values are scanned in ascending order.
INDEX	RANGE SCAN DESCENDING	Retrieval of one or more rowids from an index. Indexed values are scanned in descending order.

Table 12–3 (Cont.) OPERATION and OPTIONS Values Produced by EXPLAIN PLAN

Operation	Option	Description
INDEX	FULL SCAN	Retrieval of all rowids from an index when there is no start or stop key. Indexed values are scanned in ascending order.
INDEX	FULL SCAN DESCENDING	Retrieval of all rowids from an index when there is no start or stop key. Indexed values are scanned in descending order.
INDEX	FAST FULL SCAN	Retrieval of all rowids (and column values) using multiblock reads. No sorting order can be defined. Compares to a full table scan on only the indexed columns. Only available with the cost based optimizer.
INDEX	SKIP SCAN	Retrieval of rowids from a concatenated index without using the leading column(s) in the index. Introduced in Oracle9i. Only available with the cost based optimizer.
INLIST ITERATOR	.	Iterates over the next operation in the plan 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 (These are join operations.)	.	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.
MERGE JOIN	OUTER	Merge join operation to perform an outer join statement.
MERGE JOIN	ANTI	Merge antijoin.
MERGE JOIN	SEMI	Merge semijoin.
MERGE JOIN	CARTESIAN	Can result from 1 or more of the tables not having any join conditions to any other tables in the statement. Can occur even with a join and it may not be flagged as CARTESIAN in the plan.
CONNECT BY	.	Retrieval of rows in hierarchical order for a query containing a CONNECT BY clause.
MAT_VIEW REWITE ACCESS (These are access methods.)	FULL	Retrieval of all rows from a materialized view.
MAT_VIEW REWITE ACCESS	SAMPLE	Retrieval of sampled rows from a materialized view.
MAT_VIEW REWITE ACCESS	CLUSTER	Retrieval of rows from a materialized view based on a value of an indexed cluster key.
MAT_VIEW REWITE ACCESS	HASH	Retrieval of rows from materialized view based on hash cluster key value.
MAT_VIEW REWITE ACCESS	BY ROWID RANGE	Retrieval of rows from a materialized view based on a rowid range.
MAT_VIEW REWITE ACCESS	SAMPLE BY ROWID RANGE	Retrieval of sampled rows from a materialized view based on a rowid range.
MAT_VIEW REWITE ACCESS	BY USER ROWID	If the materialized view rows are located using user-supplied rowids.
MAT_VIEW REWITE ACCESS	BY INDEX ROWID	If the materialized view is nonpartitioned and rows are located using index(es).
MAT_VIEW REWITE ACCESS	BY GLOBAL INDEX ROWID	If the materialized view is partitioned and rows are located using only global indexes.

Table 12–3 (Cont.) OPERATION and OPTIONS Values Produced by EXPLAIN PLAN

Operation	Option	Description
MAT_VIEW REWITE ACCESS	BY LOCAL INDEX ROWID	If the materialized view 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 MAT_VIEW REWRITE 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 REMOVE_LOCATION (MAT_VIEW REWRITE ACCESS only), and INVALID.
MINUS	.	Operation accepting two sets of rows and returning rows appearing in the first set but not in the second, eliminating duplicates.
NESTED LOOPS (These are join operations.)	.	Operation accepting two sets of rows, an outer set and an inner set. Oracle compares each row of the outer set with each row of the inner set, returning rows that satisfy a condition. This join method is useful for joining small subsets of data (OLTP). The join condition is an efficient way of accessing the second table.
NESTED LOOPS	OUTER	Nested loops operation to perform an outer join statement.
PARTITION	.	Iterates over the next operation in the plan 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 12–1 for valid values of partition start/stop.
PARTITION	SINGLE	Access one partition.
PARTITION	ITERATOR	Access many partitions (a subset).
PARTITION	ALL	Access all partitions.
PARTITION	INLIST	Similar to iterator, but based on an IN-list predicate.
PARTITION	INVALID	Indicates that the partition set to be accessed is empty.
PX ITERATOR	BLOCK, CHUNK	Implements the division of an object into block or chunk ranges among a set of parallel slaves
PX COORDINATOR	.	Implements the Query Coordinator which controls, schedules, and executes the parallel plan below it using parallel query slaves. It also represents a serialization point, as the end of the part of the plan executed in parallel and always has a PX SEND QC operation below it.
PX PARTITION	.	Same semantics as the regular PARTITION operation except that it appears in a parallel plan
PX RECEIVE	.	Shows the consumer/receiver slave node reading repartitioned data from a send/producer (QC or slave) executing on a PX SEND node. This information was formerly displayed into the DISTRIBUTION column. See Table 12–2 on page 12-21.

Table 12–3 (Cont.) OPERATION and OPTIONS Values Produced by EXPLAIN PLAN

Operation	Option	Description
PX SEND	QC (RANDOM), HASH, RANGE	Implements the distribution method taking place between two parallel set of slaves. Shows the boundary between two slave sets and how data is repartitioned on the send/producer side (QC or side. This information was formerly displayed into the DISTRIBUTION column. See Table 12–2 on page 12-21.
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.
SORT	UNIQUE	Operation sorting a set of rows to eliminate duplicates.
SORT	GROUP BY	Operation sorting a set of rows into groups for a query with a GROUP BY clause.
SORT	JOIN	Operation sorting a set of rows before a merge-join.
SORT	ORDER BY	Operation sorting a set of rows for a query with an ORDER BY clause.
TABLE ACCESS (These are access methods.)	FULL	Retrieval of all rows from a table.
TABLE ACCESS	SAMPLE	Retrieval of sampled rows from a table.
TABLE ACCESS	CLUSTER	Retrieval of rows from a table based on a value of an indexed cluster key.
TABLE ACCESS	HASH	Retrieval of rows from table based on hash cluster key value.
TABLE ACCESS	BY ROWID RANGE	Retrieval of rows from a table based on a rowid range.
TABLE ACCESS	SAMPLE BY ROWID RANGE	Retrieval of sampled rows from a table based on a rowid range.
TABLE ACCESS	BY USER ROWID	If the table rows are located using user-supplied rowids.
TABLE ACCESS	BY INDEX ROWID	If the table is nonpartitioned and rows are located using index(es).
TABLE ACCESS	BY GLOBAL INDEX ROWID	If the table is partitioned and rows are located using only global indexes.
TABLE ACCESS	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 REMOVE_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: *Oracle Database Reference* for more information on `PLAN_TABLE`

13

Managing Optimizer Statistics

This chapter explains why statistics are important for the query optimizer and how to gather and use optimizer statistics with the `DBMS_STATS` package.

The chapter contains the following sections:

Understanding Statistics

Optimizer statistics are a collection of data that describe more details about the database and the objects in the database. These statistics are used by the query optimizer to choose the best execution plan for each SQL statement. Optimizer statistics 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)
  - Extended statistics
- Index statistics
  - Number of leaf blocks
  - Levels
  - Clustering factor
- System statistics

- I/O performance and utilization
- CPU performance and utilization

> **Note:** The statistics mentioned in this section are optimizer statistics, which are created for the purposes of query optimization and are stored in the data dictionary. These statistics should not be confused with performance statistics visible through V$ views.

The optimizer statistics are stored in the data dictionary. They can be viewed using data dictionary views. See "Viewing Statistics" on page 13-22.

Because the objects in a database can be constantly changing, statistics must be regularly updated so that they accurately describe these database objects. Statistics are maintained automatically by Oracle or you can maintain the optimizer statistics manually using the DBMS_STATS package. For a description of the automatic and manual processes, see "Automatic Optimizer Statistics Collection" on page 13-2 or "Manual Statistics Gathering" on page 13-5.

The DBMS_STATS package also provides procedures for managing statistics. You can save and restore copies of statistics. You can export statistics from one system and import those statistics into another system. For example, you could export statistics from a production system to a test system. In addition, you can lock statistics to prevent those statistics from changing. The lock methods are described in "Locking Statistics for a Table or Schema" on page 13-19.

Automatic Optimizer Statistics Collection

The recommended approach to gathering optimizer statistics is to allow Oracle Database to automatically gather the statistics. Oracle Database gathers optimizer statistics on all database objects automatically and maintains those statistics as an automated maintenance task. The automated maintenance tasks infrastructure (known as AutoTask) schedules tasks to run automatically in Oracle Scheduler windows known as maintenance windows. By default, one window is scheduled for each day of the week. You can customize attributes of these maintenance windows, including start and end time, frequency, and days of the week. AutoTask schedules statistics gathering as an automated maintenance task in the maintenance windows to automatically collect optimizer statistics for all schema objects in the database for which there are no statistics or only stale statistics. This process is called automatic optimizer statistics collection.

Automatic optimizer statistics collection gathers optimizer statistics by calling the DBMS_STATS.GATHER_DATABASE_STATS_JOB_PROC procedure. The GATHER_DATABASE_STATS_JOB_PROC procedure collects statistics on database objects when the object has no previously gathered statistics or the existing statistics are stale because the underlying object has been modified significantly (more than 10% of the rows).The DBMS_STATS.GATHER_DATABASE_STATS_JOB_PROC is an internal procedure, but it operates in a very similar fashion to the DBMS_STATS.GATHER_DATABASE_STATS procedure using the GATHER AUTO option. The primary difference is that the DBMS_STATS.GATHER_DATABASE_STATS_JOB_PROC procedure prioritizes the database objects that require statistics, so that those objects which most need updated statistics are processed first. This ensures that the most-needed statistics are gathered before the maintenance window closes.

Automatic optimizer statistics collection eliminates many of the manual tasks associated with managing the query optimizer, and significantly reduces the risks of generating poor execution plans due to missing or stale statistics.

This section contains the following topics:

- Enabling Automatic Optimizer Statistics Collection
- Considerations When Gathering Statistics

Enabling Automatic Optimizer Statistics Collection

Optimizer statistics are automatically gathered by automatic optimizer statistics collection, which gathers statistics on all objects in the database which have stale or missing statistics. Automatic optimizer statistics collection runs as part of the automated maintenance tasks infrastructure (AutoTask) and is enabled by default to run in all predefined maintenance windows.

If for some reason automatic optimizer statistics collection is disabled, you can enable it using the `ENABLE` procedure in the `DBMS_AUTO_TASK_ADMIN` package:

```
BEGIN
  DBMS_AUTO_TASK_ADMIN.ENABLE(
    client_name => 'auto optimizer stats collection',
    operation => NULL,
    window_name => NULL);
END;
/
```

In situations when you want to disable automatic optimizer statistics collection, you can disable it using the DISABLE procedure in the DBMS_AUTO_TASK_ADMIN package:

```
BEGIN
  DBMS_AUTO_TASK_ADMIN.DISABLE(
    client_name => 'auto optimizer stats collection',
    operation => NULL,
    window_name => NULL);
END;
/
```

Automatic optimizer statistics collection relies on the modification monitoring feature, described in "Determining Stale Statistics" on page 13-12. If this feature is disabled, then the automatic optimizer statistics collection job will not be able to detect stale statistics. This feature is enabled when the `STATISTICS_LEVEL` parameter is set to `TYPICAL` or `ALL`. `TYPICAL` is the default value.

See Also:

- *Oracle Database Administrator's Guide* for information about the AutoTask infrastructure
- *Oracle Database PL/SQL Packages and Types Reference* for information about the `DBMS_AUTO_TASK_ADMIN` package

Considerations When Gathering Statistics

This section discusses:

- When to Use Manual Statistics
- Restoring Previous Versions of Statistics

- Locking Statistics

When to Use Manual Statistics

Automatic optimizer statistics collection should be sufficient for most database objects which are being modified at a moderate speed. However, there are cases where automatic optimizer statistics collection may not be adequate. Because the automatic optimizer statistics collection runs during maintenance windows, the statistics on tables which are significantly modified throughout the day may become stale. There are typically two types of such objects:

- Volatile tables that are being deleted or truncated and rebuilt during the course of the day.
- Objects which are the target of large bulk loads which add 10% or more to the object's total size.

For highly volatile tables, there are two approaches:

- The statistics on these tables can be set to NULL. When Oracle encounters a table with no statistics, Oracle dynamically gathers the necessary statistics as part of query optimization. This dynamic sampling feature is controlled by the `OPTIMIZER_DYNAMIC_SAMPLING` parameter, and this parameter should be set to a value of 2 or higher. The default value is 2. The statistics can be set to NULL by deleting and then locking the statistics:

  ```
  BEGIN
    DBMS_STATS.DELETE_TABLE_STATS('OE','ORDERS');
    DBMS_STATS.LOCK_TABLE_STATS('OE','ORDERS');
  END;
  /
  ```

  See "Dynamic Sampling Levels" on page 13-21 for information about the sampling levels that can be set.

- The statistics on these tables can be set to values that represent the typical state of the table. You should gather statistics on the table when the tables has a representative number of rows, and then lock the statistics.

  This may be more effective than automatic optimizer statistic collection, because any statistics generated on the table during the overnight batch window may not be the most appropriate statistics for the daytime workload.

For tables which are being bulk-loaded, the statistics-gathering procedures should be run on those tables immediately following the load process, preferably as part of the same script or job that is running the bulk load.

For external tables, statistics are not collected during `GATHER_SCHEMA_STATS`, `GATHER_DATABASE_STATS`, and automatic optimizer statistics collection processing. However, you can collect statistics on an individual external table using `GATHER_TABLE_STATS`. Sampling on external tables is not supported so the `ESTIMATE_PERCENT` option should be explicitly set to `NULL`. Because data manipulation is not allowed against external tables, it is sufficient to analyze external tables when the corresponding file changes.

If the monitoring feature is disabled by setting `STATISTICS_LEVEL` to `BASIC`, automatic optimizer statistics collection cannot detect stale statistics. In this case statistics need to be manually gathered. See "Determining Stale Statistics" on page 13-12 for information on the automatic monitoring facility.

Another area in which statistics need to be manually gathered are the system statistics. These statistics are not automatically gathered. See "System Statistics" on page 13-13 for more information.

Statistics on fixed objects, such as the dynamic performance tables, need to be manually collected using GATHER_FIXED_OBJECTS_STATS procedure. Fixed objects record current database activity; statistics gathering should be done when database has representative activity.

Restoring Previous Versions of Statistics

Whenever statistics in dictionary are modified, old versions of statistics are saved automatically for future restoring. Statistics can be restored using RESTORE procedures of DBMS_STATS package. See "Restoring Previous Versions of Statistics" on page 13-17 for more information.

Locking Statistics

In some cases, you may want to prevent any new statistics from being gathered on a table or schema by the DBMS_STATS_JOB process, such as highly volatile tables discussed in "When to Use Manual Statistics" on page 13-4. In those cases, the DBMS_STATS package provides procedures for locking the statistics for a table or schema. See "Locking Statistics for a Table or Schema" on page 13-19 for more information.

Manual Statistics Gathering

If you choose not to use automatic optimizer statistics collection, then you need to manually collect statistics in all schemas, including system schemas. If the data in your database changes regularly, you also need to gather statistics regularly to ensure that the statistics accurately represent characteristics of your database objects.

Gathering Statistics with DBMS_STATS Procedures

Statistics are gathered using the DBMS_STATS package. This PL/SQL package is also used to modify, view, export, import, and delete statistics.

Note: Do not use the COMPUTE and ESTIMATE clauses of ANALYZE statement to collect optimizer statistics. These clauses are supported solely for backward compatibility and may be removed in a future release. The DBMS_STATS package collects a broader, more accurate set of statistics, and gathers statistics more efficiently.

You may continue to use ANALYZE statement for other purposes not related to optimizer statistics collection:

- To use the VALIDATE or LIST CHAINED ROWS clauses
- To collect information on free list blocks

The DBMS_STATS package can gather statistics on table and indexes, as well as individual columns and partitions of tables. It does not gather cluster statistics; however, you can use DBMS_STATS to gather statistics on the individual tables instead of the whole cluster.

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. The older statistics are saved and can be restored later if necessary. See "Restoring Previous Versions of Statistics" on page 13-17.

When gathering statistics on system schemas, you can use the procedure `DBMS_STATS.GATHER_DICTIONARY_STATS`. This procedure gathers statistics for all system schemas, including `SYS` and `SYSTEM`, and other optional schemas, such as `CTXSYS` and `DRSYS`.

When statistics are updated for a database object, Oracle invalidates any currently parsed SQL statements that access the object. The next time such a statement executes, the statement is re-parsed and the optimizer automatically chooses a new execution plan based on the new statistics. Distributed statements accessing objects with new statistics on remote databases are not invalidated. The new statistics take effect the next time the SQL statement is parsed.

Table 13–1 lists the procedures in the `DBMS_STATS` package for gathering statistics on database objects:

Table 13–1 Statistics Gathering Procedures in the DBMS_STATS Package

Procedure	Collects
`GATHER_INDEX_STATS`	Index statistics
`GATHER_TABLE_STATS`	Table, column, and index statistics
`GATHER_SCHEMA_STATS`	Statistics for all objects in a schema
`GATHER_DICTIONARY_STATS`	Statistics for all dictionary objects
`GATHER_DATABASE_STATS`	Statistics for all objects in a database

> **See Also:** *Oracle Database PL/SQL Packages and Types Reference* for syntax and examples of all `DBMS_STATS` procedures

When using any of these procedures, there are several important considerations for statistics gathering:

- Statistics Gathering Using Sampling
- Parallel Statistics Gathering
- Statistics on Partitioned Objects
- Column Statistics and Histograms
- Extended Statistics
- Determining Stale Statistics
- User-defined Statistics

Statistics Gathering Using Sampling

The statistics-gathering operations can utilize sampling to estimate statistics. Sampling is an important technique for gathering statistics. Gathering statistics without sampling requires full table scans and sorts of entire tables. Sampling minimizes the resources necessary to gather statistics.

Sampling is specified using the `ESTIMATE_PERCENT` argument to the `DBMS_STATS` procedures. While the sampling percentage can be set to any value, Oracle 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. AUTO_SAMPLE_SIZE lets Oracle determine the best sample size necessary for good statistics, based on the statistical property of the object. Because each type of statistics has different requirements, the size of the actual sample taken may not be the same across the table, columns, or indexes. For example, to collect table and column statistics for all tables in the OE schema with auto-sampling, you could use:

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

When the ESTIMATE_PERCENT parameter is manually specified, the DBMS_STATS gathering procedures may automatically increase the sampling percentage if the specified percentage did not produce a large enough sample. This ensures the stability of the estimated values by reducing fluctuations.

Parallel Statistics Gathering

The statistics-gathering operations can run either serially or in parallel. The degree of parallelism can be specified with the DEGREE argument to the DBMS_STATS gathering procedures. Parallel statistics gathering can be used in conjunction with sampling. Oracle recommends setting the DEGREE parameter to DBMS_STATS.AUTO_DEGREE. This setting allows Oracle to choose an appropriate degree of parallelism based on the size of the object and the settings for the parallel-related init.ora parameters.

Note that certain types of index statistics are not gathered in parallel, including cluster indexes, domain indexes, and bitmap join indexes.

Statistics on Partitioned Objects

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. The type of partitioning statistics to be gathered is specified in the GRANULARITY argument to the DBMS_STATS gathering procedures.

Depending on the SQL statement being optimized, the optimizer can choose to use either the partition (or subpartition) statistics or the global statistics. Both types of statistics are important for most applications, and Oracle recommends setting the GRANULARITY parameter to AUTO to gather both types of partition statistics.

With partitioned tables, new data is usually loaded into a new partition. As new partitions are added and data loaded, statistics need to be gathered on the new partition, and global statistics need to be kept up to date. If the INCREMENTAL value for a partition table is set to TRUE, and you gather statistics on that table with the GRANULARITY parameter set to AUTO, Oracle will gather statistics on the new partition and update the global table statistics by scanning only those partitions that have been modified and not the entire table. If the INCREMENTAL value for the partitioned table is set to FALSE (default value), then a full table scan is used to maintain the global statistics. This is a highly resource intensive and time consuming operation for large tables.

Note: When you set INCREMENTAL to TRUE for a partitioned table, the SYSAUX tablespace consumes additional space to maintain the global statistics.

Use the DBMS_STATS.SET_TABLE_PREF procedure to change the INCREMENTAL value for a partition table. For more information, see *Oracle Database PL/SQL Packages and Types Reference*.

Column Statistics and Histograms

When gathering statistics on a table, DBMS_STATS gathers information about the data distribution of the columns within the table. The most basic information about the data distribution is the maximum value and minimum value of the column. However, this level of statistics may be insufficient for the optimizer's needs if the data within the column is skewed. For skewed data distributions, histograms can also be created as part of the column statistics to describe the data distribution of a given column. Histograms are described in more details in "Viewing Histograms" on page 13-23.

Histograms are specified using the METHOD_OPT argument of the DBMS_STATS gathering procedures. Oracle recommends setting the METHOD_OPT to FOR ALL COLUMNS SIZE AUTO. With this setting, Oracle automatically determines which columns require histograms and the number of buckets (size) of each histogram. You can also manually specify which columns should have histograms and the size of each histogram.

Note: If you need to remove all rows from a table when using DBMS_STATS, use TRUNCATE instead of dropping and re-creating the same table. When a table is dropped, workload information used by the auto-histogram gathering feature and saved statistics history used by the RESTORE_*_STATS procedures will be lost. Without this data, these features will not function properly.

Extended Statistics

Oracle can also gather statistics on a group of columns within a table or an expression on a column. For more details on these, refer to:

- MultiColumn Statistics
- Expression Statistics

MultiColumn Statistics

When multiple columns from a single table are used together in the where clause of a query (multiple single column predicates), the relationship between the columns can strongly affect the combined selectivity for the column group.

For example, consider the customers table in the SH schema. The columns cust_state_province and country_id are related, with cust_state_province determining the country_id for each customer. Querying the customers table where the cust_state_province is California:

```
Select count(*)
from sh.customers
where cust_state_province = 'CA';
```

returns the following value:

```
 COUNT(*)
----------
    3341
```

Adding an extra predicate on the country_id column does not change the result when the country_id is 52790 (United States of America).

```
Select count(*)
from customers
where cust_state_province = 'CA'
and country_id=52790;
```

returns the same value as the previous query:

```
 COUNT(*)
----------
    3341
```

However, if the country_id has any other value, such as 52775 (Brazil), as in the following query:

```
Select count(*)
from customers
where cust_state_province = 'CA'
and     country_id=52775;
```

then the returned value is:

```
 COUNT(*)
----------
       0
```

With individual column statistics the optimizer has no way of knowing that the cust_state_province and the country_id columns are related. By gathering statistics on these columns as a group (column group), the optimizer will now have a more accurate selectivity value for the group, instead of having to generate the value based on the individual column statistics.

By default, Oracle creates column groups for a table, based on the workload analysis, similar to how it is done for histograms.

You can also create column groups manually by using the DBMS_STATS package. You can use this package to create a column group, get the name of a column group, or delete a column group from a table.

Creating a Column Group Use the create_extended_statistics function to create a column group. The create_extended_statistics function returns the system-generated name of the newly created column group. Table 13–2 lists the input parameters for this function:

Table 13–2 Parameters for the create_extended_statistics Function

Parameter	Description
owner	Schema owner. NULL indicates current schema.
tab_name	Name of the table to which the column group is being added.
extension	Columns in the column group.

For example, if you wish to add a column group consisting of the cust_state_province and country_id columns to the customers table in SH schema:

```
declare
  cg_name varchar2(30);
begin
 cg_name := dbms_stats.create_extended_stats(null,'customers',
'(cust_state_province',country_id)');
end;
/
```

Getting a Column Group Use the show_extended_stats_name function to obtain the name of the column group for a given set of columns. Table 13–3 lists the input parameters for this function:

Table 13–3 Parameters for the show_extended_stats_name Function

Parameter	Description
owner	Schema owner. NULL indicates current schema.
tab_name	Name of the table to which the column group belongs.
extension	Name of the column group.

For example, use the following query to obtain the column group name for a set of columns on the customers table:

```
select
sys.dbms_stats.show_extended_stats_name('sh','customers','(cust_state_province,cou
ntry_id)') col_group_name from dual;
```

You will obtain an output similar to the following:

```
COL_GROUP_NAME
----------------
SYS_STU#S#WF25Z#QAHIHE#MOFFMM
```

Dropping a Column Group Use the drop_extended_stats function to delete a column group from a table. Table 13–4 lists the input parameters for this function:

Table 13–4 Parameters for the drop_extended_stats Function

Parameter	Description
owner	Schema owner. NULL indicates current schema.
tab_name	Name of the table to which the column group belongs.
extension	Name of the column group to be deleted.

For example, the following statement deletes a column group from the customers table:

```
exec
dbms_stats.drop_extended_stats('sh','customers','(cust_state_province,country_id)'
);
```

Monitoring Column Groups Use the dictionary table user_stat_extensions to obtain information about MultiColumn statistics:

```
Select extension_name, extension
```

```
from user_stat_extensions
where table_name='CUSTOMERS';

EXTENSION_NAME                      EXTENSION
--------------------------------------------------------------------------
SYS_STU#S#WF25Z#QAHIHE#MOFFMM_     ("CUST_STATE_PROVINCE","COUNTRY_ID")
```

Use the following query to find the number of distinct values and find whether a histogram has been created for a column group:

```
select e.extension col_group, t.num_distinct, t.histogram
  2  from user_stat_extensions e, user_tab_col_statistics t
  3  where e.extension_name=t.column_name
  4  and t.table_name='CUSTOMERS';

COL_GROUP                              NUM_DISTINCT           HISTOGRAM
-------------------------------------------------------------------------------
("COUNTRY_ID","CUST_STATE_PROVINCE")  145                    FREQUENCY
```

Gathering Statistics on Column Groups The `METHOD_OPT` argument of the `DBMS_STATS` package enables you to gather statistics on column groups. If you set the value of this argument to `FOR ALL COLUMNS SIZE AUTO`, the optimizer will gather statistics on all the existing column groups. To collect statistics on a new column group, specify the group using `FOR COLUMNS`. The column group will be automatically created as part of statistic gathering.

For example, the following statement creates a new column group for the `customers` table on the columns `cust_state_province`, `country_id` and gathers statistics (including histograms) on the entire table and the new column group:

```
Exec dbms_stats.gather_table_stats(null,'customers',method_opt =>
'for all columns size skewonly
for columns (cust_state_province,country_id) skewonly');
```

> **Note:** The optimizer will only use MultiColumn statistics with equality predicates.

Expression Statistics

When a function is applied to a column in the `where` clause of a query (`function(col1)=constant`), the optimizer has no way of knowing how that function will affect the selectivity of the column. By gathering expression statistics on the expression `function(col1)`, the optimizer will have a more accurate selectivity value.

An example of such a function is:

```
Select count(*)
From customers
Where lower(cust_state_province)='ca';
```

Creating Expression Statistics You can create statistics on an expression as part of the `gather_table_stats` procedure:

```
exec dbms_stats.gather_table_stats(null,'customers', method_opt =>
'for all columns size skewonly
 for columns (lower(cust_state_province)) skewonly');
```

You can also use the create_extended_statistics function to accomplish this:

```
select
dbms_stats.create_extended_stats(null,'customers','(lower(cust_state_province))')
from dual;
```

Monitoring Expression Statistics Use the dictionary table user_stat_extensions to obtain information about expression statistics:

```
Select extension_name, extension
from user_stat_extensions
where table_name='CUSTOMERS';

EXTENSION_NAME                    EXTENSION
--------------------------------------------------------------------------
SYS_STUBPHJSBRKOIK9O2YV3W8HOUE    (LOWER("CUST_STATE_PROVINCE"))
```

Use the following query to find the number of distinct values and find whether a histogram has been created:

```
select e.extension col_group, t.num_distinct, t.histogram
  2  from user_stat_extensions e, user_tab_col_statistics t
  3  where e.extension_name=t.column_name
  4  and t.table_name='CUSTOMERS';

COL_GROUP                         NUM_DISTINCT          HISTOGRAM
--------------------------------------------------------------------------
(LOWER("CUST_STATE_PROVINCE"))    145                   FREQUENCY
```

Dropping Expression Statistics Use the drop_extended_stats function to delete expression statistics from a table:

```
exec dbms_stats.drop_extended_stats(null,'customers','(lower(country_id))');
```

Determining Stale Statistics

Statistics must be regularly gathered on database objects as those database objects are modified over time. In order to determine whether or not a given database object needs new database statistics, Oracle provides a table monitoring facility. This monitoring is enabled by default when STATISTICS_LEVEL is set to TYPICAL or ALL. Monitoring tracks the approximate number of INSERTs, UPDATEs, and DELETEs for that table, as well as whether the table has been truncated, since the last time statistics were gathered. The information about changes of tables can be viewed in the USER_TAB_MODIFICATIONS view. Following a data-modification, there may be a few minutes delay while Oracle propagates the information to this view. Use the DBMS_STATS.FLUSH_DATABASE_MONITORING_INFO procedure to immediately reflect the outstanding monitored information kept in the memory.

The GATHER_DATABASE_STATS or GATHER_SCHEMA_STATS procedures gather new statistics for tables with stale statistics when the OPTIONS parameter is set to GATHER STALE or GATHER AUTO. If a monitored table has been modified more than 10%, then these statistics are considered stale and gathered again.

User-defined Statistics

You can create user-defined optimizer statistics to support user-defined indexes and functions. When you associate a statistics type with a column or domain index, Oracle calls the statistics collection method in the statistics type whenever statistics are gathered for database objects.

You should gather new column statistics on a table after creating a function-based index, to allow Oracle to collect column statistics equivalent information for the expression. This is done by calling the statistics-gathering procedure with the METHOD_OPT argument set to FOR ALL HIDDEN COLUMNS.

> **See Also:** *Oracle Database Data Cartridge Developer's Guide* for details about implementing user-defined statistics

When to Gather Statistics

When gathering statistics manually, you not only need to determine how to gather statistics, but also when and how often to gather new statistics.

For an application in which tables are being incrementally modified, you may only need to gather new statistics every week or every month. The simplest way to gather statistics in these environment is to use a script or job scheduling tool to regularly run the GATHER_SCHEMA_STATS and GATHER_DATABASE_STATS procedures. 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.

For tables which are being substantially modified in batch operations, such as with bulk loads, statistics should be gathered on those tables as part of the batch operation. The DBMS_STATS procedure should be called as soon as the load operation completes.

For partitioned tables, there are often cases in which only a single partition is modified. In those cases, statistics can be gathered only on those partitions rather than gathering statistics for the entire table. However, gathering global statistics for the partitioned table may still be necessary.

> **See Also:** *Oracle Database PL/SQL Packages and Types Reference* for more information about the GATHER_SCHEMA_STATS and GATHER_DATABASE_STATS procedures in the DBMS_STATS package

System Statistics

System statistics describe the system's hardware characteristics, such as I/O and CPU performance and utilization, to the query optimizer. When choosing an execution plan, the optimizer estimates the I/O and CPU resources required for each query. System statistics enable the query optimizer to more accurately estimate I/O and CPU costs, enabling the query optimizer to choose a better execution plan.

When Oracle gathers system statistics, it analyzes system activity in a specified time period (workload statistics) or simulates a workload (noworkload statistics). The statistics are collected using the DBMS_STATS.GATHER_SYSTEM_STATS procedure. Oracle highly recommends that you gather system statistics.

Note: You must have DBA privileges or GATHER_SYSTEM_STATISTICS role to update dictionary system statistics.

Table 13–5 lists the optimizer system statistics gathered by the DBMS_STATS package and the options for gathering or manually setting specific system statistics.

Table 13–5 Optimizer System Statistics in the DBMS_STAT Package

Parameter Name	Description	Initialization	Options for Gathering or Setting Statistics	Unit
cpuspeedNW	Represents noworkload CPU speed. CPU speed is the average number of CPU cycles in each second.	At system startup	Set gathering_mode = NOWORKLOAD or set statistics manually.	Millions/sec.
ioseektim	I/O seek time equals seek time + latency time + operating system overhead time.	At system startup 10 (default)	Set gathering_mode = NOWORKLOAD or set statistics manually.	ms
iotfrspeed	I/O transfer speed is the rate at which an Oracle database can read data in the single read request.	At system startup 4096 (default)	Set gathering_mode = NOWORKLOAD or set statistics manually.	Bytes/ms
cpuspeed	Represents workload CPU speed. CPU speed is the average number of CPU cycles in each second.	None	Set gathering_mode = NOWORKLOAD, INTERVAL, or START|STOP, or set statistics manually.	Millions/sec.
maxthr	Maximum I/O throughput is the maximum throughput that the I/O subsystem can deliver.	None	Set gathering_mode = NOWORKLOAD, INTERVAL, or START|STOP, or set statistics manually.	Bytes/sec.
slavethr	Slave I/O throughput is the average parallel slave I/O throughput.	None	Set gathering_mode = INTERVAL or START|STOP, or set statistics manually.	Bytes/sec.
sreadtim	Single block read time is the average time to read a single block randomly.	None	Set gathering_mode = INTERVAL or START|STOP, or set statistics manually.	ms
mreadtim	Multiblock read is the average time to read a multiblock sequentially.	None	Set gathering_mode = INTERVAL or START|STOP, or set statistics manually.	ms
mbrc	Multiblock count is the average multiblock read count sequentially.	None	Set gathering_mode = INTERVAL or START|STOP, or set statistics manually.	blocks

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 offers two options for gathering system statistics:

- Workload Statistics
- Noworkload Statistics

These options better facilitate the gathering process to the physical database and workload: when workload system statistics are gathered, noworkload system statistics will be ignored. Noworkload system statistics are initialized to default values at the first database startup.

> **See Also:** *Oracle Database PL/SQL Packages and Types Reference* for detailed information on the procedures in the DBMS_STATS package for implementing system statistics

Workload Statistics

Workload statistics, introduced in Oracle 9*i*, gather single and multiblock read times, mbrc, CPU speed (cpuspeed), maximum system throughput, and average slave throughput. The sreadtim, mreadtim, and mbrc are computed by comparing the number of physical sequential and random reads between two points in time from the beginning to the end of a workload. These values are implemented through counters that change when the buffer cache completes synchronous read requests. Since the counters are in the buffer cache, they include not only I/O delays, but also waits related to latch contention and task switching. Workload statistics thus depend on the activity the system had during the workload window. If system is I/O bound—both

latch contention and I/O throughput—it will be reflected in the statistics and will therefore promote a less I/O intensive plan after the statistics are used. Furthermore, workload statistics gathering does not generate additional overhead.

In release 9.2, maximum I/O throughput and average slave throughput were added to set a lower limit for a full table scan (FTS).

Gathering Workload Statistics

To gather workload statistics, either:

- Run the `dbms_stats.gather_system_stats('start')` procedure at the beginning of the workload window, then the `dbms_stats.gather_system_stats('stop')` procedure at the end of the workload window.
- Run `dbms_stats.gather_system_stats('interval', interval=>N)` where `N` is the number of minutes when statistics gathering will be stopped automatically.

To delete system statistics, run `dbms_stats.delete_system_stats()`. Workload statistics will be deleted and reset to the default noworkload statistics.

Multiblock Read Count

In release 10.2, the optimizer uses the value of `mbrc` when performing full table scans (FTS). The value of `db_file_multiblock_read_count` is set to the maximum allowed by the operating system by default. However, the optimizer uses `mbrc=8` for costing. The "real" `mbrc` is actually somewhere in between since serial multiblock read requests are processed by the buffer cache and split in two or more requests if some blocks are already pinned in the buffer cache, or when the segment size is smaller than the read size. The `mbrc` value gathered as part of workload statistics is thus useful for FTS estimation.

During the gathering process of workload statistics, it is possible that `mbrc` and `mreadtim` will not be gathered if no table scans are performed during serial workloads, as is often the case with OLTP systems. On the other hand, FTS occur frequently on DSS systems but may run parallel and bypass the buffer cache. In such cases, `sreadtim` will still be gathered since index lookup are performed using the buffer cache. If Oracle cannot gather or validate gathered `mbrc` or `mreadtim`, but has gathered `sreadtim` and `cpuspeed`, then only `sreadtim` and `cpuspeed` will be used for costing. FTS cost will be computed using analytical algorithm implemented in previous releases. Another alternative to computing `mbrc` and `mreadtim` is to force FTS in serial mode to allow the optimizer to gather the data.

Noworkload Statistics

Noworkload statistics consist of I/O transfer speed, I/O seek time, and CPU speed (`cpuspeednw`). The major difference between workload statistics and noworkload statistics lies in the gathering method.

Noworkload statistics gather data by submitting random reads against all data files, while workload statistics uses counters updated when database activity occurs. `isseektim` represents the time it takes to position the disk head to read data. Its value usually varies from 5 ms to 15 ms, depending on disk rotation speed and the disk or RAID specification. The I/O transfer speed represents the speed at which one operating system process can read data from the I/O subsystem. Its value varies greatly, from a few MBs per second to hundreds of MBs per second. Oracle uses relatively conservative default settings for I/O transfer speed.

In Oracle 10*g*, Oracle uses noworkload statistics and the CPU cost model by default. The values of noworkload statistics are initialized to defaults at the first instance startup:

```
ioseektim = 10ms
iotrfspeed = 4096 bytes/ms
cpuspeednw = gathered value, varies based on system
```

If workload statistics are gathered, noworkload statistics will be ignored and Oracle will use workload statistics instead.

Gathering Noworkload Statistics

To gather noworkload statistics, run `dbms_stats.gather_system_stats()` with no arguments. There will be an overhead on the I/O system during the gathering process of noworkload statistics. The gathering process may take from a few seconds to several minutes, depending on I/O performance and database size.

The information is analyzed and verified for consistency. In some cases, the value of noworkload statistics may remain its default value. In such cases, repeat the statistics gathering process or set the value manually to values that the I/O system has according to its specifications by using the `dbms_stats.set_system_stats` procedure.

Managing Statistics

This section discusses:

- Pending Statistics
- Restoring Previous Versions of Statistics
- Exporting and Importing Statistics
- Restoring Statistics Versus Importing or Exporting Statistics
- Locking Statistics for a Table or Schema
- Setting Statistics
- Handling Missing Statistics

Pending Statistics

Starting with the 11*g* Release 1 (11.1), when gathering statistics, you have the option to automatically publish the statistics at the end of the gather operation (default behavior), or to have the new statistics saved as pending. Saving the new statistics as pending allows you to validate the new statistics and publish them only if they are satisfactory.

You can check whether or not the statistics will be automatically published as soon as they are gathered by checking the value of the `PUBLISH` attribute using the `DBMS_STATS` package:

```
Select dbms_stats.get_prefs('PUBLISH') publish from dual;
```

This query will return either `TRUE` or `FALSE`. `TRUE` indicates that the statistics will be published as and when they are gathered, while `FALSE` indicates that the statistics will be kept pending.

Note: Published statistics are stored in data dictionary views, such as USER_TAB_STATS and USER_IND_STATS. Pending statistics are stored in views such as USER_TAB_PENDING_STATS and USER_IND_PENDING_STATS.

You can change the PUBLISH setting at either the schema or the table level. For example, to change the PUBLISH setting for the customers table in the SH schema, execute the statement:

```
Exec dbms_stats.set_table_prefs('SH', 'CUSTOMERS', 'PUBLISH', 'false');
```

Subsequently, when you gather statistics on the customers table, the statistics will not be automatically published when the gather job completes. Instead, the newly gathered statistics will be stored in the USER_TAB_PENDING_STATS table.

By default, the optimizer uses the published statistics stored in the data dictionary views. If you want the optimizer to use the newly collected pending statistics, set the initialization parameter OPTIMIZER_PENDING_STATISTICS to TRUE (the default value is FALSE), and run a workload against the table or schema:

```
alter session set optimizer_pending_statistics = TRUE;
```

The optimizer will use the pending statistics instead of the published statistics when compiling SQL statements.If the pending statistics are valid, they can be made public by executing the following statement:

```
Exec dbms_stats.publish_pending_stats(null, null);
```

You can also publish the pending statistics for a specific database object. For example, by using the following statement:

```
Exec dbms_stats.publish_pending_stats('SH','CUSTOMERS');
```

If you do not want to publish the pending statistics, delete them by executing the following statement:

```
Exec dbms_stats.delete_pending_stats('SH','CUSTOMERS');
```

You can export pending statistics using dbms_stats.export_pending_stats function. Exporting pending statistics to a test system enables you to run a full workload against the new statistics.

Restoring Previous Versions of Statistics

Whenever statistics in dictionary are modified, old versions of statistics are saved automatically for future restoration. Statistics can be restored using RESTORE procedures of DBMS_STATS package. These procedures use a time stamp as an argument and restore statistics as of that time stamp. This is useful in case newly collected statistics leads to some sub-optimal execution plans and the administrator wants to revert to the previous set of statistics.

There are dictionary views that display the time of statistics modifications. These views are useful in determining the time stamp to be used for statistics restoration.

- Catalog view DBA_OPTSTAT_OPERATIONS contain history of statistics operations performed at schema and database level using DBMS_STATS.
- The views *_TAB_STATS_HISTORY views (ALL, DBA, or USER) contain a history of table statistics modifications.

The old statistics are purged automatically at regular intervals based on the statistics history retention setting and the time of the recent analysis of the system. Retention is configurable using the ALTER_STATS_HISTORY_RETENTION procedure of DBMS_STATS. The default value is 31 days, which means that you would be able to restore the optimizer statistics to any time in last 31 days.

Automatic purging is enabled when STATISTICS_LEVEL parameter is set to TYPICAL or ALL. If automatic purging is disabled, the old versions of statistics need to be purged manually using the PURGE_STATS procedure.

The other DBMS_STATS procedures related to restoring and purging statistics include:

- PURGE_STATS: This procedure can be used to manually purge old versions beyond a time stamp.
- GET_STATS_HISTORY_RETENTION: This function can be used to get the current statistics history retention value.
- GET_STATS_HISTORY_AVAILABILTY: This function gets the oldest time stamp where statistics history is available. Users cannot restore statistics to a time stamp older than the oldest time stamp.

When restoring previous versions of statistics, the following limitations apply:

- RESTORE procedures cannot restore user-defined statistics.
- Old versions of statistics are not stored when the ANALYZE command has been used for collecting statistics.

> **Note:** If you need to remove all rows from a table when using DBMS_STATS, use TRUNCATE instead of dropping and re-creating the same table. When a table is dropped, workload information used by the auto-histogram gathering feature and saved statistics history used by the RESTORE_*_STATS procedures will be lost. Without this data, these features will not function properly.

Exporting and Importing Statistics

Statistics can be exported and imported from the data dictionary to user-owned tables. This enables you to create multiple versions of statistics for the same schema. It also enables you to copy statistics from one database to another database. You may want to do this to copy the statistics from a production database to a scaled-down test database.

> **Note:** Exporting and importing statistics is a distinct concept from the EXP and IMP utilities of the database. The DBMS_STATS export and import packages do utilize IMP and EXP dump files.

Before exporting statistics, you first need to create a table for holding the statistics. This statistics table is created using the procedure DBMS_STATS.CREATE_STAT_TABLE. After this table is created, then you can export statistics from the data dictionary into your statistics table using the DBMS_STATS.EXPORT_*_STATS procedures. The statistics can then be imported using the DBMS_STATS.IMPORT_*_STATS procedures.

Note that the optimizer does not use statistics stored in a user-owned table. The only statistics used by the optimizer are the statistics stored in the data dictionary. In order

to have the optimizer use the statistics in a user-owned tables, you must import those statistics into the data dictionary using the statistics import procedures.

In order to move statistics from one database to another, you must first export the statistics on the first database, then copy the statistics table to the second database, using the EXP and IMP utilities or other mechanisms, and finally import the statistics into the second database.

Note: The EXP and IMP utilities export and import optimizer statistics from the database along with the table. One exception is that statistics are not exported with the data if a table has columns with system-generated names.

Restoring Statistics Versus Importing or Exporting Statistics

The functionality for restoring statistics is similar in some respects to the functionality of importing and exporting statistics. In general, you should use the restore capability when:

- You want to recover older versions of the statistics. For example, to restore the optimizer behavior to an earlier date.
- You want the database to manage the retention and purging of statistics histories.

You should use EXPORT/IMPORT_*_STATS procedures when:

- You want to experiment with multiple sets of statistics and change the values back and forth.
- You want to move the statistics from one database to another database. For example, moving statistics from a production system to a test system.
- You want to preserve a known set of statistics for a longer period of time than the desired retention date for restoring statistics.

Locking Statistics for a Table or Schema

Statistics for a table or schema can be locked. Once statistics are locked, no modifications can be made to those statistics until the statistics have been unlocked. These locking procedures are useful in a static environment in which you want to guarantee that the statistics never change.

The DBMS_STATS package provides two procedures for locking and two procedures for unlocking statistics:

- LOCK_SCHEMA_STATS
- LOCK_TABLE_STATS
- UNLOCK_SCHEMA_STATS
- UNLOCK_TABLE_STATS

Setting Statistics

You can set table, column, index, and system statistics using the SET_*_STATISTICS procedures. Setting statistics in the manner is not recommended, because inaccurate or inconsistent statistics can lead to poor performance.

Estimating Statistics with Dynamic Sampling

The purpose of dynamic sampling is to improve server performance by determining more accurate estimates for predicate selectivity and statistics for tables and indexes. The statistics for tables and indexes include table block counts, applicable index block counts, table cardinalities, and relevant join column statistics. These more accurate estimates allow the optimizer to produce better performing plans.

You can use dynamic sampling to:

- Estimate single-table predicate selectivities when collected statistics cannot be used or are likely to lead to significant errors in estimation.
- Estimate statistics for tables and relevant indexes without statistics.
- Estimate statistics for tables and relevant indexes whose statistics are too out of date to trust.

This dynamic sampling feature is controlled by the `OPTIMIZER_DYNAMIC_SAMPLING` parameter. For dynamic sampling to automatically gather the necessary statistics, this parameter should be set to a value of 2 or higher. The default value is 2. See "Dynamic Sampling Levels" on page 13-21 for information about the sampling levels that can be set.

How Dynamic Sampling Works

The primary performance attribute is compile time. Oracle determines at compile time whether a query would benefit from dynamic sampling. If so, a recursive SQL statement is issued to scan a small random sample of the table's blocks, and to apply the relevant single table predicates to estimate predicate selectivities. The sample cardinality can also be used, in some cases, to estimate table cardinality. Any relevant column and index statistics are also collected.

Depending on the value of the `OPTIMIZER_DYNAMIC_SAMPLING` initialization parameter, a certain number of blocks are read by the dynamic sampling query.

> **See Also:** *Oracle Database Reference* for details about this initialization parameter

When to Use Dynamic Sampling

For a query that normally completes quickly (in less than a few seconds), you will not want to incur the cost of dynamic sampling. However, dynamic sampling can be beneficial under any of the following conditions:

- A better plan can be found using dynamic sampling.
- The sampling time is a small fraction of total execution time for the query.
- The query will be executed many times.

Dynamic sampling can be applied to a subset of a single table's predicates and combined with standard selectivity estimates of predicates for which dynamic sampling is not done.

How to Use Dynamic Sampling to Improve Performance

You control dynamic sampling with the `OPTIMIZER_DYNAMIC_SAMPLING` parameter, which can be set to a value from `0` to `10`. The default is `2`.

- A value of `0` means dynamic sampling will not be done.

- Increasing the value of the parameter results in more aggressive application of dynamic sampling, in terms of both the type of tables sampled (analyzed or unanalyzed) and the amount of I/O spent on sampling.

Dynamic sampling is repeatable if no rows have been inserted, deleted, or updated in the table being sampled. The parameter `OPTIMIZER_FEATURES_ENABLE` turns off dynamic sampling if set to a version prior to 9.2.0.

Dynamic Sampling Levels

The sampling levels are as follows if the dynamic sampling level used is from a cursor hint or from the `OPTIMIZER_DYNAMIC_SAMPLING` initialization parameter:

- Level 0: Do not use dynamic sampling.
- Level 1: Sample all tables that have not been analyzed if the following criteria are met: (1) there is at least 1 unanalyzed table in the query; (2) this unanalyzed table is joined to another table or appears in a subquery or non-mergeable view; (3) this unanalyzed table has no indexes; (4) this unanalyzed table has more blocks than the number of blocks that would be used for dynamic sampling of this table. The number of blocks sampled is the default number of dynamic sampling blocks (32).
- Level 2: Apply dynamic sampling to all unanalyzed tables. The number of blocks sampled is two times the default number of dynamic sampling blocks.
- Level 3: Apply dynamic sampling to all tables that meet Level 2 criteria, plus all tables for which standard selectivity estimation used a guess for some predicate that is a potential dynamic sampling predicate. The number of blocks sampled is the default number of dynamic sampling blocks. For unanalyzed tables, the number of blocks sampled is two times the default number of dynamic sampling blocks.
- Level 4: Apply dynamic sampling to all tables that meet Level 3 criteria, plus all tables that have single-table predicates that reference 2 or more columns. The number of blocks sampled is the default number of dynamic sampling blocks. For unanalyzed tables, the number of blocks sampled is two times the default number of dynamic sampling blocks.
- Levels 5, 6, 7, 8, and 9: Apply dynamic sampling to all tables that meet the previous level criteria using 2, 4, 8, 32, or 128 times the default number of dynamic sampling blocks respectively.
- Level 10: Apply dynamic sampling to all tables that meet the Level 9 criteria using all blocks in the table.

The sampling levels are as follows if the dynamic sampling level for a table is set using the DYNAMIC_SAMPLING optimizer hint:

- Level 0: Do not use dynamic sampling.
- Level 1: The number of blocks sampled is the default number of dynamic sampling blocks (32).
- Levels 2, 3, 4, 5, 6, 7, 8, and 9: The number of blocks sampled is 2, 4, 8, 16, 32, 64, 128, or 256 times the default number of dynamic sampling blocks respectively.
- Level 10: Read all blocks in the table.

> **See Also:** *Oracle Database SQL Reference* for information about setting the sampling levels with the `DYNAMIC_SAMPLING` hint

Handling Missing Statistics

When Oracle encounters a table with missing statistics, Oracle dynamically gathers the necessary statistics needed by the optimizer. However, for certain types of tables, Oracle does not perform dynamic sampling. These include remote tables and external tables. In those cases and also when dynamic sampling has been disabled, the optimizer uses default values for its statistics, shown in Table 13–6 and Table 13–7.

Table 13–6 Default Table Values When Statistics Are Missing

Table Statistic	Default Value Used by Optimizer
`Cardinality`	num_of_blocks * (block_size - cache_layer) / avg_row_len
`Average row length`	100 bytes
`Number of blocks`	100 or actual value based on the extent map
`Remote cardinality`	2000 rows
`Remote average row length`	100 bytes

Table 13–7 Default Index Values When Statistics Are Missing

Index Statistic	Default Value Used by Optimizer
`Levels`	1
`Leaf blocks`	25
`Leaf blocks/key`	1
`Data blocks/key`	1
`Distinct keys`	100
`Clustering factor`	800

Viewing Statistics

This section discusses:

- Statistics on Tables, Indexes and Columns
- Viewing Histograms

Statistics on Tables, Indexes and Columns

Statistics on tables, indexes, and columns are stored in the data dictionary. To view statistics in the data dictionary, query the appropriate data dictionary view (`USER`, `ALL`, or `DBA`). These `DBA_*` views include the following:

- DBA_TABLES
- DBA_OBJECT_TABLES
- DBA_TAB_STATISTICS
- DBA_TAB_COL_STATISTICS
- DBA_TAB_HISTOGRAMS
- DBA_TAB_COLS
- DBA_COL_GROUP_COLUMNS
- DBA_INDEXES

- DBA_IND_STATISTICS
- DBA_CLUSTERS
- DBA_TAB_PARTITIONS
- DBA_TAB_SUBPARTITIONS
- DBA_IND_PARTITIONS
- DBA_IND_SUBPARTITIONS
- DBA_PART_COL_STATISTICS
- DBA_PART_HISTOGRAMS
- DBA_SUBPART_COL_STATISTICS
- DBA_SUBPART_HISTOGRAMS

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

Viewing Histograms

Column statistics may be stored as histograms. These histograms provide accurate estimates of the distribution of column data. Histograms provide improved selectivity estimates in the presence of data skew, resulting in optimal execution plans with nonuniform data distributions.

Oracle uses the following types of histograms for column statistics:

- Height-Balanced Histograms
- Frequency Histograms

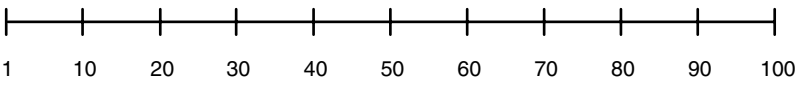
The type of histogram is stored in the `HISTOGRAM` column of the `*TAB_COL_STATISTICS` views (`USER` and `DBA`). This column can have values of `HEIGHT BALANCED`, `FREQUENCY`, or `NONE`.

Height-Balanced Histograms

In a height-balanced histogram, the column values are divided into bands so that each band contains approximately the same number of rows. The useful information that the histogram provides is where in the range of values the endpoints fall.

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 similar to Figure 13–1, where the numbers are the endpoint values.

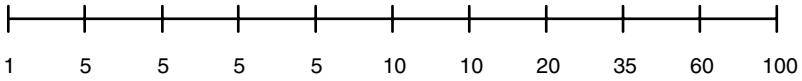
Figure 13–1 Height-Balanced Histogram with Uniform Distribution



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 that are between 60 and 100 in this example of uniform distribution.

If the data is not uniformly distributed, then the histogram might look similar to Figure 13–2.

Figure 13–2 Height-Balanced Histogram with Non-Uniform Distribution



In this case, most of the rows have the value 5 for the column. Only 1/10 of the rows have values between 60 and 100.

Height-balanced histograms can be viewed using the `*TAB_HISTOGRAMS` tables, as shown in Example 13–1.

Example 13–1 Viewing Height-Balanced Histogram Statistics

```
BEGIN
  DBMS_STATS.GATHER_table_STATS (OWNNAME => 'OE', TABNAME => 'INVENTORIES',
  METHOD_OPT => 'FOR COLUMNS SIZE 10 quantity_on_hand');
END;
/

SELECT column_name, num_distinct, num_buckets, histogram
  FROM USER_TAB_COL_STATISTICS
 WHERE table_name = 'INVENTORIES' AND column_name = 'QUANTITY_ON_HAND';

COLUMN_NAME                    NUM_DISTINCT NUM_BUCKETS HISTOGRAM
------------------------------ ------------ ----------- ---------------
QUANTITY_ON_HAND                        237          10 HEIGHT BALANCED

SELECT endpoint_number, endpoint_value
  FROM USER_HISTOGRAMS
 WHERE table_name = 'INVENTORIES' and column_name = 'QUANTITY_ON_HAND'
  ORDER BY endpoint_number;

ENDPOINT_NUMBER ENDPOINT_VALUE
--------------- --------------
              0              0
              1             27
              2             42
              3             57
              4             74
              5             98
              6            123
              7            149
              8            175
              9            202
             10            353
```

In the query output, one row corresponds to one bucket in the histogram.

Frequency Histograms

In a frequency histogram, each value of the column corresponds to a single bucket of the histogram. Each bucket contains the number of occurrences of that single value. Frequency histograms are automatically created instead of height-balanced histograms when the number of distinct values is less than or equal to the number of histogram buckets specified. Frequency histograms can be viewed using the `*TAB_HISTOGRAMS` tables, as shown in Example 13–2.

Example 13–2 Viewing Frequency Histogram Statistics

```
BEGIN
```

```
  DBMS_STATS.GATHER_table_STATS (OWNNAME => 'OE', TABNAME => 'INVENTORIES',
  METHOD_OPT => 'FOR COLUMNS SIZE 20 warehouse_id');
END;
/

SELECT column_name, num_distinct, num_buckets, histogram
  FROM USER_TAB_COL_STATISTICS
 WHERE table_name = 'INVENTORIES' AND column_name = 'WAREHOUSE_ID';

COLUMN_NAME                    NUM_DISTINCT NUM_BUCKETS HISTOGRAM
------------------------------ ------------ ----------- ---------------
WAREHOUSE_ID                              9           9 FREQUENCY

SELECT endpoint_number, endpoint_value
  FROM USER_HISTOGRAMS
 WHERE table_name = 'INVENTORIES' and column_name = 'WAREHOUSE_ID'
  ORDER BY endpoint_number;

ENDPOINT_NUMBER ENDPOINT_VALUE
--------------- --------------
             36              1
            213              2
            261              3
            370              4
            484              5
            692              6
            798              7
            984              8
           1112              9
```

14

Using Indexes and Clusters

This chapter provides an overview of data access methods using indexes and clusters that can enhance or degrade performance.

The chapter contains the following sections:

Understanding Index Performance

This section describes the following:

Tuning the Logical Structure

Although query 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. Make sure you are monitoring a representative workload to avoid dropping an index which is used, but not by the workload you sampled.

Also, 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 from being taken out on a child table.

See Also:

- *Oracle Database SQL Reference* for information on the `ALTER INDEX MONITORING USAGE` statement
- *Oracle Database Application Developer's Guide - Fundamentals* for information on foreign keys

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 parsed, 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.

Note that creating an index to tune one statement can affect the optimizer's choice of execution plans for other statements. 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 execution plans, and rerun the SQL trace facility after you have tuned those statements that you initially identified for tuning.

Index Tuning using the SQLAccess Advisor

The SQLAccess Advisor is an alternative to manually determining which indexes are required. This advisor recommends a set of indexes when invoked from **Advisor Central** in Oracle Enterprise Manager or run through the `DBMS_ADVISOR` package APIs. The SQLAccess Advisor either recommends using a workload or it generates a hypothetical workload for a specified schema. Various workload sources are available, such as the current contents of the SQL Cache, a user defined set of SQL statements, or a SQL Tuning set. Given a workload, the SQLAccess Advisor generates a set of recommendations from which you can select the indexes that are to be implemented. An implementation script is provided which can be executed manually or automatically through Oracle Enterprise Manager. For information on the SQLAccess Advisor, see "Overview of the SQL Access Advisor" on page 18-1.

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 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 for Performance" on page 14-11.
- Choose 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 other values.
- 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 the index is modified frequently, as in a high concurrency OLTP application.
- 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: *Oracle Database 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 11-20.

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 table1(x, y, z);
```

- `x`, `xy`, and `xyz` combinations of columns are leading portions of the index
- `yz`, `y`, and `z` combinations of columns are *not* leading portions of the index

Choosing Keys for Composite Indexes

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.

Ordering Keys for Composite Indexes

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 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 query optimizer 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 the following methods:

- Use the `NO_INDEX` hint to give the query optimizer maximum flexibility while disallowing the use of a certain index.
- Use the `FULL` hint to instruct the optimizer to choose a full table scan instead of an index scan.
- Use the `INDEX` or `INDEX_COMBINE` hints to instruct the optimizer to use one index or a set of listed indexes instead of another.

> **See Also:** Chapter 19, "Using 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 for each index entry), then it might be better to use sequential index lookup rather 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.

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:** *Oracle Database 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 by using the `ALTER INDEX` statement with the `COALESCE` option. This option lets you combine leaf levels of an index to free blocks for reuse. You can also rebuild the index online.

> **See Also:** *Oracle Database SQL Reference* and *Oracle 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. By placing a constraint in the enabled novalidated state, you 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);
```

Now you can use Import or Fast Loader to load data into table 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;
```

At this point, users can start performing INSERTs, UPDATEs, DELETEs, and SELECTs on table t.

```
ALTER TABLE t ENABLE CONSTRAINT apk;
ALTER TABLE x ENABLE CONSTRAINT afk;
```

Now the constraints are enabled and validated.

> **See Also:** *Oracle Database Concepts* for a complete discussion of integrity constraints

Using Function-based Indexes for Performance

A function-based index includes columns that are either transformed by a function, such as the UPPER function, or included in an expression, such as col1 + col2. With a function-based index, you can store computation-intensive expressions in the index.

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. This allows Oracle to bypass computing the value of the expression when processing SELECT and DELETE statements. 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.

Oracle treats descending indexes as function-based indexes. The columns marked DESC are sorted in descending order.

For example, function-based indexes defined with the UPPER(*column_name*) or LOWER(*column_name*) keywords allow case-insensitive searches. The index created in the following statement:

```
CREATE INDEX uppercase_idx ON employees (UPPER(last_name));
```

facilitates processing queries such as:

```
SELECT * FROM employees
    WHERE UPPER(last_name) = 'MARKSON';
```

See Also:

- *Oracle Database Application Developer's Guide - Fundamentals* and *Oracle Database Administrator's Guide* for more information on using function-based indexes
- *Oracle Database SQL Reference* for more information on the `CREATE INDEX` statement

Using Partitioned Indexes for Performance

Similar to partitioned tables, partitioned indexes improve manageability, availability, performance, and scalability. They can either be partitioned independently (global indexes) or automatically linked to a table's partitioning method (local indexes).

Oracle supports both range and hash partitioned global indexes. In a range partitioned global index, each index partition contains values defined by a partition bound. In a hash partitioned global index, each partition contains values determined by the Oracle hash function.

The hash method can improve performance of indexes where a small number leaf blocks in the index have high contention in multiuser OLTP environment. In some OLTP applications, index insertions happen only at the right edge of the index. This could happen when the index is defined on monotonically increasing columns. In such situations right edge of the index becomes a hotspot because of contention for index pages, buffers, latches for update, and additional index maintenance activity, which results in performance degradation.

With hash partitioned global indexes index entries are hashed to different partitions based on partitioning key and the number of partitions. This spreads out contention over number of defined partitions, resulting in increased throughput.
Hash-partitioned global indexes would benefit TPC-H refresh functions that are executed as massive PDMLs into huge fact tables because contention for buffer latches would be spread out over multiple partitions.

With hash partitioning, an index entry will be mapped to a particular index partition based on the hash value generated by Oracle. The syntax to create hash-partitioned global index is very similar to hash-partitioned table. Queries involving equality and `IN` predicates on index partitioning key can efficiently use global hash partitioned index to answer queries quickly.

See Also: *Oracle Database Concepts* and *Oracle Database Administrator's Guide* for more information on global indexes tables

Using Index-Organized Tables for Performance

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.

Global hash-partitioned indexes are supported for index-organized tables and can provide performance benefits in a multiuser OLTP environment.

See Also: *Oracle Database Concepts* and *Oracle Database Administrator's Guide* for more information on index-organized tables

Using Bitmap Indexes for Performance

Bitmap indexes can substantially improve performance of queries that have all of 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.
- The bitmap indexes used in the queries have been created on some or all of these low- or medium-cardinality columns.
- The tables in the queries 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:** *Oracle Database Concepts* and *Oracle Database Data Warehousing Guide* for more information on bitmap indexing

Using Bitmap Join Indexes for Performance

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-saving way to reduce 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(s) of the dimension tables and the foreign key column(s) 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:** *Oracle Database Data Warehousing Guide* for examples and restrictions of bitmap join indexes

Using Domain Indexes for Performance

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 the following:

- 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` statement.

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

Using Clusters for Performance

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

To create a cluster, use the `CREATE CLUSTER` statement.

> **See Also:** *Oracle 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 for each cluster key value varies significantly. This causes waste of space for the low cardinality key value; it causes 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.

> **See Also:** *Oracle Database Administrator's Guide* for more information on creating clusters

Using Hash Clusters for Performance

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.

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.
- Use sorted hash clusters, where rows corresponding to each value of the hash function are sorted on a specific columns in ascending order, when response time can be improved on operations with this sorted clustered data.
- Do not store a table in a hash cluster if the application often performs full table scans and if 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 reduces 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, as long as hashing is appropriate for the table based on the considerations in this list.

> **See Also:**
>
> - *Oracle Database Administrator's Guide* for information on managing hash clusters
> - *Oracle Database SQL Reference* for information on the `CREATE CLUSTER` statement

15

Using SQL Plan Management

This chapter describes how to manage SQL execution plans using SQL plan management. SQL plan management prevents performance regressions resulting from sudden changes to the execution plan of a SQL statement by providing components for capturing, selecting, and evolving SQL plan information.

The execution plan of a SQL statement can be affected by various changes, such as:

- New optimizer version
- Changes to optimizer statistics and optimizer parameters
- Changes to schema and metadata definitions
- Changes to system settings
- SQL profile creating

Some events may cause an irreversible change to an execution plan, such as dropping an index. These changes can cause regressions in SQL performance, and fixing them manually can be difficult and time consuming.

The SQL tuning features of Oracle Database generate SQL profiles that help the optimizer in producing well-tuned plans, but this is a reactive mechanism and cannot guarantee stable performance when drastic changes happen to the system. SQL tuning can only resolve performance issues after they have occurred and are identified. For example, a SQL statement may become a high-load statement due to a plan change, but this cannot be resolved by SQL tuning until after the plan change occurs.

SQL plan management is a preventative mechanism that records and evaluates the execution plans of SQL statements over time, and builds SQL plan baselines composed of a set of existing plans known to be efficient. The SQL plan baselines are then used to preserve performance of corresponding SQL statements, regardless of changes occurring in the system.

Common usage scenarios where SQL plan management can improve or preserve SQL performance include:

- A database upgrade that installs a new optimizer version usually results in plan changes for a small percentage of SQL statements, with most of the plan changes resulting in either no performance change or improvement. However, certain plan changes may cause performance regressions. The use of SQL plan baselines significantly minimizes potential performance regressions resulting from a database upgrade.
- Ongoing system and data changes can impact plans for some SQL statements, potentially causing performance regressions. The use of SQL plan baselines will help to minimize performance regressions and stabilize SQL performance.

- Deployment of new application modules means introducing new SQL statements into the system. The application software may use appropriate SQL execution plans developed under a standard test configuration for the new SQL statements. If your system configuration is significant different from the test configuration, the SQL plan baselines can be evolved over time to produce better performance.

This chapter contains the following topics:

Managing SQL Plan Baselines

Managing SQL plan baselines involves three phases:

Capturing SQL Plan Baselines

During the SQL plan baseline capture phase, Oracle Database records information about SQL statement execution to detect plan changes and decide whether it is safe to use new plans. To do so, Oracle Database maintains a history of plans for individual SQL statements. Since ad-hoc SQL statements do not repeat and thus do not suffer performance degradation, plan history is maintained only for repeatable SQL statements.

To recognize repeatable SQL statements, a statement log is maintained that contains identifiers of various SQL statements the optimizer has evaluated over time. A SQL statement is recognized as repeatable when it is parsed or executed again after it has been logged.

For each SQL statement, the system maintains a plan history that contains all plans generated by the optimizer. However, before a plan in the plan history can be made acceptable for use by the optimizer, the plan must be verified to not cause performance regression. The set of all accepted plans in the plan history is the SQL plan baseline.

The SQL Plan Baseline Capture phase can be configured for automatic capture of plan history and SQL plan baselines for repeatable SQL statements, or a set of plans can be manually loaded as SQL plan baselines.

This section contains the following topics:

Automatic Plan Capture

When automatic plan capture is enabled, the system automatically creates and maintains the plan history for SQL statements using information provided by the optimizer. The plan history will include relevant information used by the optimizer to reproduce an execution plan, such as the SQL text, outline, bind variables, and compilation environment.

The initial plan generated for a SQL statement is marked as accepted for use by the optimizer, and represents both the plan history and the SQL plan baseline. All subsequent plans will be included in the plan history, and those plans that are verified not to cause performance regressions will be added to the SQL plan baseline during the SQL plan baseline evolution phase.

To enable automatic plan capture, set the `OPTIMIZER_CAPTURE_SQL_PLAN_BASELINES` initialization parameter to `TRUE`. By default, this parameter is set to `FALSE`.

Manual Plan Loading

Another way to create SQL plan baselines is by manually loading existing plans for a set of SQL statements as SQL plan baselines. The manually loaded plans are not verified for performance, but are added as accepted plans to existing or new SQL plan baselines. Manual plan loading can be used in conjunction with or as an alternative to automatic plan capture. Manual plan loading can be performed by:

- Loading Plans from SQL Tuning Sets and AWR Snapshots
- Loading Plans from the Cursor Cache

> **See Also:** "SQL Management Base" on page 15-8

Loading Plans from SQL Tuning Sets and AWR Snapshots To load plans from a SQL Tuning Set, use the `LOAD_PLANS_FROM_SQLSET` function of the `DBMS_SPM` package:

```
DECLARE
my_plans pls_integer;
BEGIN
  my_plans := DBMS_SPM.LOAD_PLANS_FROM_SQLSET(
    sqlset_name => 'tset1');
END;
/
```

In this example, Oracle Database loads the plans stored in SQL Tuning Set named `tset1`. For information about additional parameters used by the `LOAD_PLANS_FROM_SQLSET` function, see *Oracle Database PL/SQL Packages and Types Reference*.

To load plans from Automatic Workload Repository (AWR), load the plans stored in AWR snapshots into a SQL Tuning Set before using the `LOAD_PLANS_FROM_SQLSET` function as described in this section.

> **See Also:**
>
> - "Overview of the Automatic Workload Repository" on page 5-8
> - "SQL Tuning Sets" on page 17-14

Loading Plans from the Cursor Cache To load plans from the cursor cache, use the `LOAD_PLANS_FROM_CURSOR_CACHE` function of the `DBMS_SPM` package:

```
DECLARE
my_plans pls_integer;
```

```
BEGIN
  my_plans := DBMS_SPM.LOAD_PLANS_FROM_CURSOR_CACHE(
    sql_id => '99twu5t2dn5xd');
END;
/
```

In this example, Oracle Database loads the plans located in the cursor cache for the SQL statement identified by its `sql_id`. Plans in the cursor cache can be identified by:

- SQL identifier (`SQL_ID`)
- SQL text (`SQL_TEXT`)
- One of the following attributes:
  - `PARSING_SCHEMA_NAME`
  - `MODULE`
  - `ACTION`

  See Also: *Oracle Database PL/SQL Packages and Types Reference* for information about using the `LOAD_PLANS_FROM_CURSOR_CACHE` function

Selecting SQL Plan Baselines

During the SQL plan baseline selection phase, Oracle Database detects plan changes based on the stored plan history, and selects plans to avoid potential performance regressions for a set of SQL statements.

Each time a SQL statement is compiled, the optimizer will first use a cost-based search method to build a best-cost plan, then it will try to find a matching plan in the SQL plan baseline. If a match is found, the optimizer will proceed using this plan. Otherwise, it will evaluate the cost of each accepted plan in the SQL plan baseline and select the plan with the lowest cost. The best-cost plan found by the optimizer that does not match any plans in the plan history for the SQL statement represents a new plan, and is added as a non-accepted plan to the plan history. The new plan is not used until it is verified to not cause a performance regression. However, if a change in the system (such as a dropped index) causes all accepted plans to become non-reproducible, the optimizer will select the best-cost plan. Thus, the presence of a SQL plan baseline causes the optimizer to use conservative plan selection strategy for the SQL statement.

To enable the use of SQL plan baselines, set the `OPTIMIZER_USE_SQL_PLAN_BASELINES` initialization parameter to `TRUE`. By default, this parameter is set to `TRUE`.

Evolving SQL Plan Baselines

During the SQL plan baseline evolution phase, Oracle Database evaluates the performance of new plans and integrates plans with better performance into SQL plan baselines.

When the optimizer finds a new plan for a SQL statement, the plan is added to the plan history as a non-accepted plan. The plan can then be verified for performance relative to the SQL plan baseline performance. When a non-accepted plan is verified to not cause a performance regression, it is changed to an accepted plan and integrated into the SQL plan baseline. A successful verification of a non-accepted plan consists of comparing its performance to that of a plan selected from the SQL plan baseline and ensuring that it delivers better performance.

This section describes how to evolve SQL plan baselines and contains the following topics:

- Evolving Plans With Manual Plan Loading
- Evolving Plans With DBMS_SPM.EVOLVE_SQL_PLAN_BASELINE

Evolving Plans With Manual Plan Loading

You can evolve an existing SQL plan baseline by manually loading plans either from the cursor cache or from a SQL tuning set. When you manually load plans into a SQL plan baseline, these loaded plans are added as accepted plans.

See Also: "Manual Plan Loading" on page 15-3

Evolving Plans With DBMS_SPM.EVOLVE_SQL_PLAN_BASELINE

The PL/SQL function `DBMS_SPM.EVOLVE_SQL_PLAN_BASELINE` tries to evolve new plans added by the optimizer to the plan history of existing plan baselines. If the function can verify that the new plan performs better than a plan chosen from the corresponding SQL plan baseline, the new plan is added as an accepted plan.

The following is an example of the `DBMS_SPM.EVOLVE_SQL_PLAN_BASELINE` function:

```
SET SERVEROUTPUT ON
SET LONG 10000
DECLARE
    report clob;
BEGIN
    report := DBMS_SPM.EVOLVE_SQL_PLAN_BASELINE(
                  sql_handle => 'SYS_SQL_593bc74fca8e6738');
    DBMS_OUTPUT.PUT_LINE(report);
END;
/
```

Output:

```
REPORT
--------------------------------------------------------------------------------

--------------------------------------------------------------------------------
                     Evolve SQL Plan Baseline Report
--------------------------------------------------------------------------------

Inputs:
-------
 SQL_HANDLE = SYS_SQL_593bc74fca8e6738
 PLAN_NAME  =
 TIME_LIMIT = DBMS_SPM.AUTO_LIMIT
 VERIFY     = YES
 COMMIT     = YES

Plan: SYS_SQL_PLAN_ca8e6738a57b5fc2
-----------------------------------
 Plan was verified: Time used .07 seconds.
 Passed performance criterion: Compound improvement ratio >= 7.32.
 Plan was changed to an accepted plan.

                    Baseline Plan      Test Plan     Improv. Ratio
                    -------------      ---------     -------------
 Execution Status:        COMPLETE       COMPLETE
```

```
 Rows Processed:                40             40
 Elapsed Time(ms):              23              8              2.88
 CPU Time(ms):                  23              8              2.88
 Buffer Gets:                  450             61              7.38
 Disk Reads:                     0              0
 Direct Writes:                  0              0
 Fetches:                        0              0
 Executions:                     1              1

-------------------------------------------------------------------------------
                                 Report Summary
-------------------------------------------------------------------------------
Number of SQL plan baselines verified: 1.
Number of SQL plan baselines evolved: 1.
```

In this example, Oracle Database successfully evolved a plan for a SQL statement identified by its SQL handle. Alternatively, you can use the `DBMS_SPM.EVOLVE_SQL_PLAN_BASELINE` function to specify:

- The name of a particular plan that you want to evolve
- A list of plans to evolve
- No value

  This enables Oracle Database to evolve all non-accepted plans currently in the SQL management base.

  See Also: *Oracle Database PL/SQL Packages and Types Reference* for information about using the `DBMS_SPM.EVOLVE_SQL_PLAN_BASELINE` function

Using SQL Plan Baselines with the SQL Tuning Advisor

When tuning SQL statements with the SQL Tuning Advisor, if the advisor finds a tuned plan and verifies its performance to be better than a plan chosen from the corresponding SQL plan baseline, it makes a recommendation to accept a SQL profile. When the SQL profile is accepted, the tuned plan is added to the corresponding SQL plan baseline. However, the SQL Tuning Advisor will not verify existing unaccepted plans in the plan history.

In Oracle Database 11*g*, an automatically configured task runs the SQL Tuning Advisor during a maintenance window. This automatic SQL tuning task targets high-load SQL statements as identified by the execution performance data collected in the Automatic Workload Repository (AWR) snapshots. The SQL profile recommendations made by the SQL tuning advisor are implemented by the automatic SQL tuning task. Tuned plans are thus automatically added to the SQL plan baselines of the identified high-load SQL statements.

See Also:

- "Reactive Tuning Using the SQL Tuning Advisor" on page 17-9
- "SQL Profiles" on page 17-19
- "Overview of the Automatic Workload Repository" on page 5-8

Using Fixed SQL Plan Baselines

A SQL plan baseline is fixed if it contains at least one enabled plan whose FIXED attribute is set to YES. You can use fixed SQL plan baselines to fix the set of possible plans (usually one plan) for a SQL statement, or migrate an existing stored outline by loading the "outlined" plan as a fixed plan.

If a fixed SQL plan baseline also contains non-fixed plans, the optimizer will give preference to fixed plans over non-fixed ones. This means that the optimizer will pick the fixed plan with the least cost even though a non-fixed plan may have an even lower cost. If none of the fixed plans is reproducible, then the optimizer will pick the best non-fixed plan.

The optimizer will not add new plans to a fixed SQL plan baseline. Since new plans are not automatically added, a fixed SQL plan baseline is not evolved when DBMS_SPM.EVOLVE_SQL_PLAN_BASELINE is executed. However, a fixed SQL plan baseline can be evolved by manually loading new plans into it from the cursor cache or a SQL tuning set.

When a SQL statement with a fixed SQL plan baseline is tuned using the SQL Tuning Advisor, a SQL profile recommendation has special meaning. When the SQL profile is accepted, the tuned plan is added to the fixed SQL plan baseline as a non-fixed plan. However, as described above, the optimizer will not use the tuned plan as long as a reproducible fixed plan is present. Therefore, the benefit of SQL tuning may not be realized. To enable the use of the tuned plan, manually alter the tuned plan to a fixed plan by setting its FIXED attribute to YES.

Displaying SQL Plan Baselines

To view the plans stored in the SQL plan baseline for a given statement, use the DISPLAY_SQL_PLAN_BASELINE function of the DBMS_XPLAN package:

```
select * from table(
    dbms_xplan.display_sql_plan_baseline(
        sql_handle=>'SYS_SQL_209d10fabbedc741',
        format=>'basic'));
```

The DISPLAY_SQL_PLAN_BASELINE function displays one or more execution plans for the specified SQL statement, specified by the handle (sql_handle). Alternatively, a single plan can be displayed by supplying a plan name (plan_name). For information about additional parameters used by the DISPLAY_SQL_PLAN_BASELINE function, see *Oracle Database PL/SQL Packages and Types Reference*.

This function uses plan information stored in the SQL management base to explain and display the plans. In this example, the DISPLAY_SQL_PLAN_BASELINE function displays the execution plans for the SQL statement specified by the handle SYS_SQL_209d10fabbedc741:

```
SQL handle: SYS_SQL_209d10fabbedc741
SQL text: select cust_last_name, amount_sold from customers c,
          sales s where c.cust_id=s.cust_id and cust_year_of_birth=:yob
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
Plan name: SYS_SQL_PLAN_bbedc741a57b5fc2
Enabled: YES      Fixed: NO      Accepted: NO      Origin: AUTO-CAPTURE
--------------------------------------------------------------------------------
Plan hash value: 2776326082

--------------------------------------------------------------------------------
| Id  | Operation                                | Name                          |
```

```
--------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                          |                             |
|   1 | HASH JOIN                                 |                             |
|   2 |   TABLE ACCESS BY INDEX ROWID             | CUSTOMERS                   |
|   3 |     BITMAP CONVERSION TO ROWIDS           |                             |
|   4 |     BITMAP INDEX SINGLE VALUE             | CUSTOMERS_YOB_BIX           |
|   5 |    PARTITION RANGE ALL                    |                             |
|   6 |    TABLE ACCESS FULL                      | SALES                       |
--------------------------------------------------------------------------------

--------------------------------------------------------------------------------
Plan name: SYS_SQL_PLAN_bbedc741f554c408
Enabled: YES     Fixed: NO      Accepted: YES       Origin: MANUAL-LOAD
--------------------------------------------------------------------------------
Plan hash value: 4115973128

--------------------------------------------------------------------------------
| Id  | Operation                                 | Name                        |
--------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                          |                             |
|   1 |  NESTED LOOPS                             |                             |
|   2 |    NESTED LOOPS                           |                             |
|   3 |      TABLE ACCESS BY INDEX ROWID          | CUSTOMERS                   |
|   4 |        BITMAP CONVERSION TO ROWIDS        |                             |
|   5 |          BITMAP INDEX SINGLE VALUE        | CUSTOMERS_YOB_BIX           |
|   6 |      PARTITION RANGE                      |                             |
|   7 |       BITMAP CONVERSION TO ROWIDS         |                             |
|   8 |         BITMAP INDEX SINGLE VALUE         | SALES_CUST_BIX              |
|   9 |    TABLE ACCESS BY LOCAL INDEX ROWID      | SALES                       |
--------------------------------------------------------------------------------
```

You can also display SQL plan baseline information using a `SELECT` statement directly on the `DBA_SQL_PLAN_BASELINES` view:

```
select sql_handle, plan_name, enabled, accepted, fixed from dba_sql_plan_
baselines;

SQL_HANDLE                PLAN_NAME                      ENA  ACC    FIX
------------------------------------------------------------------------
SYS_SQL_209d10fabbedc741  SYS_SQL_PLAN_bbedc741a57b5fc2  YES  NO     NO
SYS_SQL_209d10fabbedc741  SYS_SQL_PLAN_bbedc741f554c408  YES  YES    NO
```

SQL Management Base

The SQL management base (SMB) is a part of the data dictionary that resides in the `SYSAUX` tablespace. It stores statement log, plan histories, and SQL plan baselines, as well as SQL profiles. To allow weekly purging of unused plans and logs, the SMB is configured with automatic space management enabled.

You can also add plans manually to the SMB for a set of SQL statements. This feature is especially useful when upgrading Oracle Database from a pre-11*g* version, since it helps to minimize plan regressions resulting from the use of a new optimizer version.

Because the SMB is stored entirely within the `SYSAUX` tablespace, SQL plan management and SQL tuning features will not be used if this tablespace is not available.

This section contains the following topics:

- Disk Space Usage

Disk Space Usage

Disk space used by the SQL management base is regularly checked against a limit based on the size of the SYSAUX tablespace. By default, the limit for the SMB is no more than 10% of the size of the SYSAUX tablespace. The allowable range for this limit is between 1% and 50%. A weekly background process measures the total space occupied by the SMB, and when the defined limit is exceeded, the process will generate a warning that is written to the alert log. The alerts are generated weekly until either the SMB space limit is increased, the size of the SYSAUX tablespace is increased, or the disk space used by the SMB is decreased by purging SQL management objects (SQL plan baselines or SQL profiles).

To change the percentage limit, use the CONFIGURE procedure of the DBMS_SPM package:

```
BEGIN
  DBMS_SPM.CONFIGURE(
    'space_budget_percent',30);
END;
/
```

In this example, the space limit is changed to 30%. For information about additional parameters used by the CONFIGURE procedure, see *Oracle Database PL/SQL Packages and Types Reference*.

Purging Policy

A weekly scheduled purging task manages the disk space used by SQL plan management. The task runs as an automated task in the maintenance window. Any plan that has not been used for more than 53 weeks are purged, as identified by the LAST_EXECUTED timestamp stored in the SMB for that plan. The 53-week time frame ensures plan information will be available during any yearly SQL processing activity. The unused plan retention period can range between 5 weeks and 523 weeks (a little more than 10 years).

To configure the retention period, use the CONFIGURE procedure of the DBMS_SPM PL/SQL package:

```
BEGIN
  DBMS_SPM.CONFIGURE(
    'plan_retention_weeks',105);
END;
/
```

In this example, the retention period is changed to 105 weeks. For information about additional parameters used by the CONFIGURE procedure, see *Oracle Database PL/SQL Packages and Types Reference*.

SQL Management Base Configuration Parameters

The current configuration settings for the SQL management base can be viewed using the DBA_SQL_MANAGEMENT_CONFIG view. The following query shows this information:

```
select parameter_name, parameter_value from dba_sql_management_config;
```

```
PARAMETER_NAME                 PARAMETER_VALUE
------------------------------ ---------------
SPACE_BUDGET_PERCENT                        30
PLAN_RETENTION_WEEKS                       105
```

Importing and Exporting SQL Plan Baselines

Oracle Database supports the export and import of SQL plan baselines using its import and export utilities or Oracle Data Pump. Use the `DBMS_SPM` package to define a staging table, which is then used to pack and unpack SQL plan baselines.

To import a set of SQL plan baselines from one system to another:

1. On the original system, create a staging table using the `CREATE_STGTAB_BASELINE` procedure:

   ```
   BEGIN
     DBMS_SPM.CREATE_STGTAB_BASELINE(
       table_name => 'stage1');
   END;
   /
   ```

   This example creates a staging table named `stage1`.

2. Pack the SQL plan baselines you want to export from the SQL management base into the staging table using the `PACK_STGTAB_BASELINE` function:

   ```
   DECLARE
   my_plans number;
   BEGIN
     my_plans := DBMS_SPM.PACK_STGTAB_BASELINE(
       table_name => 'stage1',
       enabled => 'yes',
       creator => 'dba1');
   END;
   /
   ```

   This example packs all enabled plan baselines created by user `dba1` into the staging table `stage1`. You can select SQL plan baselines using the plan name (`plan_name`), SQL handle (`sql_handle`), or by any other plan criteria. The `table_name` parameter is mandatory.

3. Export the staging table `stage1` into a flat file using the export command or Oracle Data Pump.

4. Transfer the flat file to the target system.

5. Import the staging table `stage1` from the flat file using the import command or Oracle Data Pump.

6. Unpack the SQL plan baselines from the staging table into the SQL management base on the target system using the `UNPACK_STGTAB_BASELINE` function:

   ```
   DECLARE
   my_plans number;
   BEGIN
     my_plans := DBMS_SPM.UNPACK_STGTAB_BASELINE(
       table_name => 'stage1',
       fixed => 'yes');
   END;
   /
   ```

This example unpacks all fixed plan baselines stored in the staging table `stage1`.

For more information about using the `DBMS_SPM` package, see *Oracle Database PL/SQL Packages and Types Reference*.

16

SQL Tuning Overview

This chapter discusses goals for tuning, how to identify high-resource SQL statements, explains what should be collected, provides tuning suggestions, and discusses how to create SQL test cases to troubleshoot problems in SQL.

This chapter contains the following sections:

See Also:

- *Oracle Database Concepts* for an overview of SQL
- *Oracle Database 2 Day DBA* for information on monitoring and tuning the database

Introduction to SQL Tuning

An important facet of database system performance tuning is the tuning of SQL statements. SQL tuning involves three basic steps:

- Identifying high load or top SQL statements that are responsible for a large share of the application workload and system resources, by reviewing past SQL execution history available in the system.
- Verifying that the execution plans produced by the query optimizer for these statements perform reasonably.
- Implementing corrective actions to generate better execution plans for poorly performing SQL statements.

These three steps are repeated until the system performance reaches a satisfactory level or no more statements can be tuned.

Goals for Tuning

The objective of tuning a system is either to reduce the response time for end users of the system, or to reduce the resources used to process the same work. You can accomplish both of these objectives in several ways:

- Reduce the Workload
- Balance the Workload
- Parallelize the Workload

Reduce the Workload

SQL tuning commonly involves 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 High-Load SQL

This section describes the steps involved in identifying and gathering data on high-load SQL statements. High-load SQL are poorly-performing, resource-intensive SQL statements that impact the performance of the Oracle database. High-load SQL statements can be identified by:

- Automatic Database Diagnostic Monitor
- Automatic SQL tuning
- Automatic Workload Repository
- `V$SQL` view
- Custom Workload
- SQL Trace

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)?
- 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. Oracle Enterprise Manager provides tools for identifying resource intensive SQL statements, generating explain plans, and evaluating SQL performance.

See Also:

- *Oracle Enterprise Manager Concepts* for information about its capabilities for monitoring and tuning SQL applications
- Chapter 17, "Automatic SQL Tuning" for information on automatic SQL tuning features

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 ordering the SQL statements by elapsed time (with highest elapsed time SQL statements 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 21, "Using Application Tracing Tools"

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 operating system and Oracle statistics at the beginning and end of that period. The minimum of Oracle statistics gathered should be file I/O (`V$FILESTAT`), system statistics (`V$SYSSTAT`), and SQL statistics (`V$SQLAREA`, `V$SQL`, or `V$SQLSTATS`, `V$SQLTEXT`, `V$SQL_PLAN`, and `V$SQL_PLAN_STATISTICS`).

See Also:

- Chapter 6, "Automatic Performance Diagnostics" for information on how to use Oracle tools to gather Oracle instance performance data
- "Real-Time SQL Monitoring" on page 10-37 for information about the `V$SQL_PLAN_MONITOR` view

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$SQLSTATS. V$SQLSTATS contains resource usage information for all SQL statements in the shared pool. The data in V$SQLSTATS should be ordered by resource usage. The most common resources are:
   - Buffer gets (V$SQLSTATS.BUFFER_GETS, for high CPU using statements)
   - Disk reads (V$SQLSTATS.DISK_READS, for high I/O statements)
   - Sorts (V$SQLSTATS.SORTS, for many sorts)

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. The total number of buffer gets in the system is available in the V$SYSSTAT table, for the statistic session logical reads.

Similarly, it is possible to apportion the percentage of disk reads a statement performs out of the total disk reads performed by the system by dividing V$SQL_STATS.DISK_READS by the value for the V$SYSSTAT statistic physical reads. The SQL sections of the Automatic Workload Repository report include this data, so you do not need to perform the percentage calculations manually.

> **See Also:** *Oracle Database Reference* for information about dynamic performance views

After you have identified the candidate SQL statements, the next stage is to gather information that is necessary 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. The information gathered includes the following:

1. Complete SQL text from V$SQLTEXT
2. Structure of the tables referenced in the SQL statement, usually by describing the table in SQL*Plus
3. Definitions of any indexes (columns, column orderings), and whether the indexes are unique or non-unique
4. Optimizer statistics for the segments (including the number of rows each table, selectivity of the index columns), including the date when the segments were last analyzed
5. Definitions of any views referred to in the SQL statement
6. Repeat steps two, three, and four for any tables referenced in the view definitions found in step five
7. Optimizer plan for the SQL statement (either from EXPLAIN PLAN, V$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 lets you 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.

Automatic SQL Tuning Features

Because the manual SQL tuning process poses many challenges to the application developer, the SQL tuning process has been automated by the automatic SQL tuning features of Oracle Database. These features are designed to work equally well for OLTP and Data Warehouse type applications:

- ADDM
- SQL Tuning Advisor
- SQL Tuning Sets
- SQL Access Advisor

See Also: Chapter 17, "Automatic SQL Tuning".

ADDM

The Automatic Database Diagnostic Monitor (ADDM) analyzes the information collected by the AWR for possible performance problems with Oracle Database, including high-load SQL statements. See "Overview of the Automatic Database Diagnostic Monitor" on page 6-1.

SQL Tuning Advisor

The SQL Tuning Advisor optimizes SQL statements that have been identified as high-load SQL statements. By default, Oracle Database automatically identifies problematic SQL statements and implements tuning recommendations using the SQL Tuning Advisor during system maintenance windows as an automated maintenance task, searching for ways to improve the execution plans of the high-load SQL statements. You can also choose to run the SQL Tuning Advisor at any time on any given SQL workload to improve performance. See "Reactive Tuning Using the SQL Tuning Advisor" on page 17-9.

SQL Tuning Sets

When multiple SQL statements are used as input to ADDM, SQL Tuning Advisor, or SQL Access Advisor, a SQL Tuning Set (STS) is constructed and stored. The STS includes the set of SQL statements along with their associated execution context and basic execution statistics. See "SQL Tuning Sets" on page 17-14.

SQL Access Advisor

In addition to the SQL Tuning Advisor, the SQL Access Advisor provides advice on materialized views, indexes, and materialized view logs. The SQL Access Advisor helps you achieve performance goals by recommending the proper set of materialized views, materialized view logs, and indexes for a given workload. In general, as the

number of materialized views and indexes and the space allocated to them is increased, query performance improves. The SQL Access Advisor considers the trade-offs between space usage and query performance, and recommends the most cost-effective configuration of new and existing materialized views and indexes. See "Using the SQL Access Advisor" on page 18-5.

Developing Efficient SQL Statements

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 query optimizer 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 real cardinality (row count) of a table to the value of `DBA_TABLES.NUM_ROWS`. Additionally, 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 (that is, the join order should reflect, where possible, going to the best not-yet-used filters).
- The join method is appropriate for the number of rows being returned. For example, nested loop joins through 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 these conditions are not optimal, then consider restructuring the SQL statement or the indexes available on the tables.

Restructuring the SQL Statements

Often, rewriting an inefficient SQL statement is easier than modifying 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;
```

When you need to use SQL functions on filters or join predicates, do not use them on the columns on which you want to have an index; rather, use them on the opposite side of the predicate, as in the following statement:

```
TO_CHAR(numcol) = varcol
```

rather than

```
varcol = TO_CHAR(numcol)
```

> **See Also:** Chapter 14, "Using 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 round-trips 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 (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 through 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 semijoins for the EXISTS case.

See Also: *Oracle Database 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 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 optimizer cost of the statement with the actual number of resources used (BUFFER_GETS, DISK_READS, CPU_TIME from V$SQLSTATS or V$SQLAREA). Situations such as data skew (without the use of histograms) can adversely affect the optimizer's estimated cost for an operation.

"Example 1: Using IN - Selective Filters in the Subquery" and "Example 2: Using EXISTS - Selective Predicate in the Parent" 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 these 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 following SQL statement uses EXISTS:

```
SELECT /* EXISTS example */
        e.employee_id, e.first_name, e.last_name, e.salary
  FROM employees e
 WHERE EXISTS (SELECT 1 FROM orders o                  /* Note 1 */
                 WHERE e.employee_id = o.sales_rep_id   /* Note 2 */
                   AND o.customer_id = 144);           /* Note 3 */
```

Notes:

- 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*.

The following plan output is the execution plan (from V$SQL_PLAN) for the preceding 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 (through 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.

The following plan output is the execution plan (from V$SQL_PLAN) for the preceding statement. The optimizer rewrites the subquery into a view, which is then joined through 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 employee_ids are then used to access the employees table through 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.

The following SQL statement uses 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 FROM orders o); /* Note 4 */
```

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.

The following plan output is the execution plan (from V$SQL_PLAN) for the preceding 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 through 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
```

The following SQL statement uses 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.

The following plan output is the execution plan (from `V$SQL_PLAN`) for the preceding 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 through 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
```

Note: An even more efficient approach is to have a concatenated index on `department_id` and `job_id`. This eliminates the need to access two indexes and reduces the resources used.

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 query optimizer. 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 instruct the optimizer about how the statement should be executed.

Hints, such as `/*+FULL */` control access paths. For example:

```
SELECT /*+ FULL(e) */ e.last_name
  FROM employees e
 WHERE e.job_id = 'CLERK';
```

See Also: Chapter 11, "The Query Optimizer" and Chapter 19, "Using 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 non-join 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 19-4

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, the following statement creates a view that lists 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;
```

The following query finds employees in a specified 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 (+);
```

In the following plan table output, note that the `emp_dept` view is instantiated:

```
-------------------------------------------------------------------------------
| 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 department_name
FROM emp_dept
WHERE department_id = 10;
```

The entire view is first instantiated by performing a join of the `employees` and `departments` tables and then aggregating the data. However, you can obtain `department_name` and `department_id` directly from the `departments` table. It is inefficient to obtain this information by querying the `emp_dept` view.

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:** *Oracle Database Data Warehousing Guide* for an explanation of the dangers with subquery unnesting

Use Caution When Performing Outer Joins to Views In the case of an outer join to a multi-table view, the query optimizer (in Release 8.1.6 and later) can drive from an outer join column, if an equality predicate is defined on it.

An outer join *within* a view is 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 or joined data from fact and possibly dimension tables.

> **See Also:** *Oracle Database 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 SQL plan baselines. Storing optimizer statistics for tables will apply to all SQL statements that refer to those tables. Storing an execution plan as a SQL plan baseline maintains the plan for set of SQL statements. If both statistics and a SQL plan baseline are available for a SQL statement, the optimizer will first use a cost-based search method to build a best-cost plan, then it will try to find a matching plan in the SQL plan baseline. If a match is found, the optimizer will proceed using this plan. Otherwise, it will evaluate the cost of each of the accepted plans in the SQL plan baseline and select the plan with the lowest cost.

See Also:

- Chapter 15, "Using SQL Plan Management"
- Chapter 13, "Managing Optimizer Statistics"

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 Needed 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:** *Oracle Database SQL Reference* for syntax on the `INSERT`, `UPDATE`, and `DELETE` statements

Modify All the Data Needed 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:** *Oracle Database Administrator's Guide* or *Oracle Database Heterogeneous Connectivity Administrator's Guide* for information on tuning distributed queries

Building SQL Test Cases

For many SQL-related problems, obtaining a reproducible test case makes it easier to resolve the problem. Starting with the 11*g* Release 1 (11.1), Oracle database contains the SQL Test Case Builder, which automates the somewhat difficult and time-consuming process of gathering and reproducing as much information as possible about a problem and the environment in which it occurred.

The objective of a SQL Test Case Builder is to capture the information pertaining to a SQL-related problem, along with the exact environment under which the problem occurred, so that the problem can be reproduced and tested on a separate Oracle

database instance. Once the test case is ready, you can upload the problem to Oracle Support to enable support personnel to reproduce and troubleshoot the problem.

The information gathered by SQL Test Case Builder includes the query being executed, table and index definitions (but not the actual data), PL/SQL functions, procedures, and packages, optimizer statistics, and initialization parameter settings.

Creating a Test Case

You can access the SQL Test Case Builder from Enterprise Manager as well as manually using the `DBMS_SQLDIAG` package.

Accessing SQL Test Case Builder from the Enterprise Manager

From Enterprise Manager, the SQL Test Case Builder is accessible only when a SQL incident occurs. A SQL-related problem is referred to as a SQL incident, and each SQL incident is identified by an incident number. You can access the SQL Test Case Builder from the `Support Workbench` page in Enterprise Manager.

You can access the `Support Workbench` page in the following ways:

- In the Database Home page of Enterprise Manager, under `Diagnostic Summary`, click on the link to `Active Incidents` (indicating the number of active incidents). This opens the `Support Workbench` page, with the incidents listed in a table.

  Or

- Click **Advisor Central** under `Related Links` to open the `Advisor Central` page. Next, click **SQL Advisors** and then **Click here to go to Support Workbench** to open the `Support Workbench` page.

From the `Support Workbench` page, to access the SQL Test Case Builder:

1. Click on an incident ID to open the problem details for the particular incident.
2. Next, click **Oracle Support** in the `Investigate and Resolve` section.
3. Click **Generate Additional Dumps and Test Cases**.
4. For a particular incident, click on the icon in the `Go To Task` column to run the SQL Test Case Builder.

   Provide a location to store the test case, and also provide a name for the output.

The output of the SQL Test Case Builder is a SQL script that contains the commands required to recreate all the necessary objects and the environment.

Accessing SQL Test Case Builder Using DBMS_SQLDIAG

You can also invoke the SQL Test Case Builder manually, using the `DBMS_SQLDIAG` package. This package consists of various subprograms for the SQL Test Case Builder, some of which are listed in Table 16–1.

Table 16–1 SQL Test Case Builder Procedures in DBMS_SQLDIAG

Procedure Name	Function
EXPORT_SQL_TESTCASE	Generate a SQL test case.
EXPORT_SQL_TESTCASE_DIR_BY_INC	Generates a SQL test case corresponding to the incident ID passed as an argument.
EXPORT_SQL_TESTCASE_DIR_BY_TXT	Generates a SQL test case corresponding to the SQL text passed as an argument.

For more information on this package and all of its procedures and parameters, see *Oracle Database PL/SQL Packages and Types Reference*.

17

Automatic SQL Tuning

This chapter discusses the automatic SQL tuning features of Oracle Database. Automatic SQL tuning automates the manual process, which is complex, repetitive, and time-consuming.

This chapter contains the following sections:

> **See Also:** *Oracle Database 2 Day + Performance Tuning Guide* for information about using the automatic SQL tuning features of Oracle Database with Oracle Enterprise Manager

Automatic Tuning Optimizer

When SQL statements are executed by the Oracle database, the query optimizer is used to generate the execution plans of the SQL statements. The query optimizer operates in two modes: a normal mode and a tuning mode.

In normal mode, the optimizer compiles the SQL and generates an execution plan. The normal mode of the optimizer generates a reasonable execution plan for the vast majority of SQL statements. Under normal mode, the optimizer operates with very strict time constraints, usually a fraction of a second, during which it must find a good execution plan.

In tuning mode, the optimizer performs additional analysis to check whether the execution plan produced under normal mode can be improved further. The output of the query optimizer is not an execution plan, but a series of actions, along with their rationale and expected benefit for producing a significantly superior plan. When running in the tuning mode, the optimizer is referred to as the Automatic Tuning Optimizer.

Under tuning mode, the optimizer can take several minutes to tune a single statement. It is both time and resource intensive to invoke the Automatic Tuning Optimizer every time a query has to be hard-parsed. The Automatic Tuning Optimizer is meant to be used for complex and high-load SQL statements that have non-trivial impact on the

entire system. The Automatic Database Diagnostic Monitor (ADDM) proactively identifies high-load SQL statements which are good candidates for SQL tuning. See Chapter 6, "Automatic Performance Diagnostics". The automatic SQL tuning feature of Oracle Database also automatically identifies problematic SQL statements and implements tuning recommendations during system maintenance windows as an automated maintenance task.

The Automatic Tuning Optimizer performs four types of tuning analysis:

- Statistics Analysis
- SQL Profiling
- Access Path Analysis
- SQL Structure Analysis

Statistics Analysis

The query optimizer relies on object statistics to generate execution plans. If these statistics are stale or missing, the optimizer does not have the necessary information it needs and can generate poor execution plans. The Automatic Tuning Optimizer checks each query object for missing or stale statistics, and produces two types of output:

- Recommendations to gather relevant statistics for objects with stale or no statistics.

  Because optimizer statistics are automatically collected and refreshed, this problem may be encountered only when automatic optimizer statistics collection has been turned off. See "Automatic Optimizer Statistics Collection" on page 13-2.

- Auxiliary information in the form of statistics for objects with no statistics, and statistic adjustment factor for objects with stale statistics.

This auxiliary information is stored in an object called a SQL Profile.

SQL Profiling

The query optimizer can sometimes produce inaccurate estimates about an attribute of a statement due to lack of information, leading to poor execution plans. Traditionally, users have corrected this problem by manually adding hints to the application code to guide the optimizer into making correct decisions. For packaged applications, changing application code is not an option and the only alternative available is to log a bug with the application vendor and wait for a fix.

Automatic SQL tuning deals with this problem with its SQL profiling capability. The Automatic Tuning Optimizer creates a profile of the SQL statement called a SQL Profile, consisting of auxiliary statistics specific to that statement. The query optimizer under normal mode makes estimates about cardinality, selectivity, and cost that can sometimes be off by a significant amount resulting in poor execution plans. SQL Profile addresses this problem by collecting additional information using sampling and partial execution techniques to verify and, if necessary, adjust these estimates.

During SQL Profiling, the Automatic Tuning Optimizer also uses execution history information of the SQL statement to appropriately set optimizer parameter settings, such as changing the `OPTIMIZER_MODE` initialization parameter setting from `ALL_ROWS` to `FIRST_ROWS` for that SQL statement.

The output of this type of analysis is a recommendation to accept the SQL Profile. A SQL Profile, once accepted, is stored persistently in the data dictionary. Note that the SQL Profile is specific to a particular query. If accepted, the optimizer under normal mode uses the information in the SQL Profile in conjunction with regular database

statistics when generating an execution plan. The availability of the additional information makes it possible to produce well-tuned plans for corresponding SQL statement without requiring any change to the application code.

The scope of a SQL Profile can be controlled by the CATEGORY profile attribute. This attribute determines which user sessions can apply the profile. You can view the CATEGORY attribute for a SQL Profile in CATEGORY column of the DBA_SQL_PROFILES view. By default, all profiles are created in the DEFAULT category. This means that all user sessions where the SQLTUNE_CATEGORY initialization parameter is set to DEFAULT can use the profile.

By altering the category of a SQL profile, you can determine which sessions are affected by the creation of a profile. For example, by setting the category of a SQL Profile to DEV, only those users sessions where the SQLTUNE_CATEGORY initialization parameter is set to DEV can use the profile. All other sessions do not have access to the SQL Profile and execution plans for SQL statements are not impacted by the SQL profile. This technique enables you to test a SQL Profile in a restricted environment before making it available to other user sessions.

> **See Also:** *Oracle Database Reference* for information on the SQLTUNE_CATEGORY initialization parameter

It is important to note that the SQL Profile does not freeze the execution plan of a SQL statement, as done by stored outlines. As tables grow or indexes are created or dropped, the execution plan can change with the same SQL Profile. The information stored in it continues to be relevant, even as the data distribution or access path of the corresponding statement change. In general, it is not necessary to refresh the SQL profiles. However, over a long period of time, its content can become outdated and may need to be regenerated. This can be done by running the SQL Tuning Advisor again on the same statement to regenerate the SQL Profile.

SQL Profiles apply to the following statement types:

- SELECT statements
- UPDATE statements
- INSERT statements (only with a SELECT clause)
- DELETE statements
- CREATE TABLE statements (only with the AS SELECT clause)
- MERGE statements (the update or insert operations)

A complete set of functions are provided for management of SQL Profiles. See "SQL Profiles" on page 17-19.

Access Path Analysis

Indexes can tremendously enhance performance of a SQL statement by reducing the need for full table scans on large tables. Effective indexing is a common tuning technique. The Automatic Tuning Optimizer also explores whether a new index can significantly enhance the performance of a query. If such an index is identified, it recommends its creation.

Because the Automatic Tuning Optimizer does not analyze how its index recommendation can affect the entire SQL workload, it also recommends running the SQL Access Advisor utility on the SQL statement along with a representative SQL workload. The SQL Access Advisor looks at the impact of creating an index on the

entire SQL workload before making any recommendations. See "Automatic SQL Tuning Features" on page 16-5.

SQL Structure Analysis

The Automatic Tuning Optimizer identifies common problems with structure of SQL statements that can lead to poor performance. These could be syntactic, semantic, or design problems with the statement. In each of these cases the Automatic Tuning Optimizer makes relevant suggestions to restructure the SQL statements. The alternative suggested is similar, but not equivalent, to the original statement.

For example, the optimizer may suggest to replace `UNION` operator with `UNION ALL` or to replace `NOT IN` with `NOT EXISTS`. An application developer can then determine if the advice is applicable to their situation or not. For instance, if the schema design is such that there is no possibility of producing duplicates, then the `UNION ALL` operator is much more efficient than the `UNION` operator. These changes require a good understanding of the data properties and should be implemented only after careful consideration.

SQL Tuning Advisor

The SQL Tuning Advisor takes one or more SQL statements as an input and invokes the Automatic Tuning Optimizer to perform SQL tuning on the statements. The output of the SQL Tuning Advisor is in the form of an advice or recommendations, along with a rationale for each recommendation and its expected benefit. The recommendation relates to collection of statistics on objects, creation of new indexes, restructuring of the SQL statement, or creation of a SQL profile. You can choose to accept the recommendation to complete the tuning of the SQL statements.

Oracle Database can automatically tune SQL statements by identifying problematic SQL statements and implementing tuning recommendations using the SQL Tuning Advisor during system maintenance windows. You can also run the SQL Tuning Advisor selectively on a single or a set of SQL statements that have been identified as problematic.

Automatic SQL Tuning Advisor

Oracle Database automatically runs the SQL Tuning Advisor on selected high-load SQL statements from the Automatic Workload Repository (AWR) that qualify as tuning candidates. This task, called Automatic SQL Tuning, runs in the default maintenance windows on a nightly basis. You can customize attributes of the maintenance windows, including start and end time, frequency, and days of the week.

> **See Also:** *Oracle Database Administrator's Guide* for information about automated maintenance task

Once automatic SQL tuning begins, which by default runs for at most one hour during a maintenance window, the following steps are performed:

1. Identify SQL candidates in the AWR for tuning.

   Oracle Database analyzes statistics in the AWR and generates a list of potential SQL statements that are eligible for tuning. These statements include repeating high-load statements that have a significant impact on the system. Only SQL statements that have an execution plan with a high potential for improvement will be tuned. Recursive SQL and statements that have been tuned recently (in the last month) are ignored, as are parallel queries, DMLs, DDLs, and SQL statements

with performance problems that are caused by concurrency issues. The SQL statements that are selected as candidates are then ordered based on their performance impact. The performance impact of a SQL statement is calculated by summing the CPU time and the I/O times captured in the AWR for that SQL statement in the past week.

2. Tune each SQL statement individually by calling the SQL Tuning Advisor.

   During the tuning process, all recommendation types are considered and reported, but only SQL profiles can be implemented automatically.

3. Test SQL profiles by executing the SQL statement.

   If a SQL profile is recommended, test the new SQL profile by executing the SQL statement both with and without the SQL profile. If the performance improvement improves at least threefold, the SQL profile will be accepted (when the `ACCEPT_SQL_PROFILES` task parameter is set to `TRUE`). Otherwise, only the recommendation to create a SQL profile will be reported in the automatic SQL tuning reports.

4. Optionally implement the SQL profiles provided they meet the criteria of threefold performance improvement.

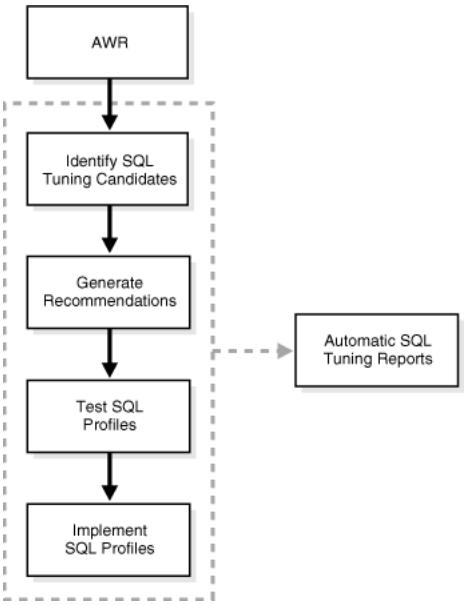
   Note that other factors are considered when deciding whether or not to implement the SQL profile. For example, a SQL profile is not implemented if the objects referenced in the SQL statement have stale optimizer statistics. You can identify which SQL profiles have been implemented automatically as their type will be set to `AUTO` in the `DBA_SQL_PROFILES` view.

   If SQL plan management is used and there is already an existing plan baseline for the SQL statement, a new plan baseline will be added when a SQL profile is created. As a result, the new and improved SQL execution plan will be used by the optimizer immediately after the SQL profile is created. For information about SQL plan management, see Chapter 15, "Using SQL Plan Management".

At any time during or after the automatic SQL tuning process, you can view the results using the automatic SQL tuning report. This report describes in detail all the SQL statements that were analyzed, the recommendations generated, as well as the SQL profiles that were automatically implemented.

Figure 17–1 illustrates the steps performed by Oracle Database during the automatic SQL tuning process.

Figure 17–1 Automatic SQL Tuning



This section contains the following topics:

- Enabling and Disabling Automatic SQL Tuning
- Configuring Automatic SQL Tuning
- Viewing Automatic SQL Tuning Reports

Enabling and Disabling Automatic SQL Tuning

Automatic SQL tuning runs as part of the automated maintenance tasks infrastructure.

To enable automatic SQL tuning, use the `ENABLE` procedure in the `DBMS_AUTO_TASK_ADMIN` package:

```
BEGIN
  DBMS_AUTO_TASK_ADMIN.ENABLE(
    client_name => 'sql tuning advisor',
    operation => NULL,
    window_name => NULL);
END;
/
```

To disable automatic SQL tuning, use the DISABLE procedure in the DBMS_AUTO_TASK_ADMIN package:

```
BEGIN
  DBMS_AUTO_TASK_ADMIN.DISABLE(
    client_name => 'sql tuning advisor',
    operation => NULL,
    window_name => NULL);
END;
/
```

You can pass a specific window name using the `window_name` parameter to enable or disable the task in certain maintenance windows only.

Setting the `STATISTICS_LEVEL` parameter to `BASIC` will disable automatic statistics gathering by the AWR and, as a result, also disable automatic SQL tuning.

See Also:

- *Oracle Database Administrator's Guide* for information about the AutoTask infrastructure
- *Oracle Database PL/SQL Packages and Types Reference* for information about the DBMS_AUTO_TASK_ADMIN package

Configuring Automatic SQL Tuning

The behavior of the automatic SQL tuning task can be configured using the DBMS_SQLTUNE package. To use the APIs, the user needs at least the ADVISOR privilege.

In addition to configuring the standard behavior of the SQL Tuning Advisor, the DBMS_SQLTUNE package enables you to configure automatic SQL tuning by specifying the task parameters using the SET_TUNING_TASK_PARAMETER procedure. Because the automatic tuning task is owned by SYS, only the SYS user can set the task parameters.

Table 17–2 lists the parameters that are specific to automatic SQL tuning which can be configured.

Table 17–1 SET_TUNING_TASK_PARAMETER Automatic SQL Tuning Parameters

Parameter	Description
ACCEPT_SQL_PROFILE	Specifies whether to accept SQL profiles automatically.
MAX_SQL_PROFILES_PER_EXEC	Specifies the limit of SQL profiles that are accepted for each automatic SQL tuning task. Consider setting the limit of SQL profiles that are accepted for each automatic SQL tuning task based on the acceptable level of changes that can be made to the system on a daily basis.
MAX_AUTO_SQL_PROFILES	Specifies the limit of SQL profiles that are accepted in total.
EXECUTION_DAYS_TO_EXPIRE	Specifies the number of days for which to save the task history in the advisor framework schema. By default, the task history is saved for 30 days before it expires.

To configure automatic SQL tuning, run the SET_TUNING_TASK_PARAMETER procedure in the DBMS_SQLTUNE package:

```
BEGIN
  DBMS_SQLTUNE.SET_TUNING_TASK_PARAMETER(
    task_name => 'SYS_AUTO_SQL_TUNING_TASK',
    parameter => 'ACCEPT_SQL_PROFILES', value => 'TRUE');
END;
/
```

In this example, the automatic SQL tuning task is configured to automatically accept SQL profiles recommended by the SQL Tuning Advisor.

See Also:

- "Configuring a SQL Tuning Task" on page 17-12 for information about other parameters that can be configured for a SQL tuning task
- *Oracle Database PL/SQL Packages and Types Reference* for information about the DBMS_SQLTUNE package

Viewing Automatic SQL Tuning Reports

The automatic SQL tuning report is generated using the DBMS_SQLTUNE.REPORT_AUTO_TUNING_TASK function and contains information about all executions of the automatic SQL tuning task. To run this report, you need the ADVISOR privilege and SELECT privileges on the DBA_ADVISOR views. Unlike the standard SQL tuning report generated using the DBMS_SQLTUNE.REPORT_TUNING_TASK function, which only contains information about a single task execution of the SQL Tuning Advisor, the automatic SQL tuning report contains information about multiple executions of the automatic SQL tuning task.

To view the automatic SQL tuning report, run the REPORT_AUTO_TUNING_TASK function in the DBMS_SQLTUNE package:

```
variable my_rept CLOB;
BEGIN
  :my_rept :=DBMS_SQLTUNE.REPORT_AUTO_TUNING_TASK(
    begin_exec => NULL,
    end_exec => NULL,
    type => 'TEXT',
    level => 'TYPICAL',
    section => 'ALL',
    object_id => NULL,
    result_limit => NULL);
END;
/

print :my_rept
```

In this example, a text report is generated to display all SQL statements that were analyzed in the most recent execution, including recommendations that were not implemented, and all sections of the report are included.

> **See Also:** *Oracle Database PL/SQL Packages and Types Reference* for information about the DBMS_SQLTUNE package

Depending on the sections that were included in the report, you can view information about the automatic SQL tuning task in the following sections of the report:

- General information

  The general information section provides a high-level description of the automatic SQL tuning task, including information about the inputs given for the report, the number of SQL statements tuned during the maintenance, and the number of SQL profiles that were created

- Summary

  The summary section lists the SQL statements (by their SQL identifiers) that were tuned during the maintenance window and the estimated benefit of each SQL profile, or their actual execution statistics after test executing the SQL statement with the SQL profile

- Tuning findings

  This section contains the following information about each SQL statement analyzed by the SQL Tuning Advisor:

  – All findings associated with each SQL statement

  – Whether the profile was accepted on the system, and why

  - Whether the SQL profile is currently enabled on the system
  - Detailed execution statistics captured when testing the SQL profile
- Explain plans

  This section shows the old and new explain plans used by each SQL statement analyzed by the SQL Tuning Advisor.
- Errors

  This section lists all errors encountered by the automatic SQL tuning task.

Reactive Tuning Using the SQL Tuning Advisor

The SQL Tuning Advisor can be invoked manually for on-demand tuning of one or more SQL statements. To tune multiple statements, you need to create a SQL tuning set (STS). A SQL tuning set is a database object that stores SQL statements along with their execution context. You can create a SQL tuning set using command line APIs or Oracle Enterprise Manager. See "SQL Tuning Sets" on page 17-14.

This section contains the following topics:

Input Sources

The input for the SQL Tuning Advisor can come from several sources. These input sources include:

- Automatic Database Diagnostic Monitor

  The primary input source is the Automatic Database Diagnostic Monitor (ADDM). By default, ADDM runs proactively once every hour and analyzes key statistics gathered by the Automatic Workload Repository (AWR) over the last hour to identify any performance problems including high-load SQL statements. If a high-load SQL is identified, ADDM recommends running SQL Tuning Advisor on the SQL. See "Overview of the Automatic Database Diagnostic Monitor" on page 6-1.
- Automatic Workload Repository

  The second most important input source is the Automatic Workload Repository (AWR). The AWR takes regular snapshots of the system activity, including high-load SQL statements ranked by relevant statistics, such as CPU consumption and wait time.

  You can view the AWR and manually identify high-load SQL statements and run the SQL Tuning Advisor on them, though this is done automatically by Oracle Database as part of the automatic SQL tuning process. By default, the AWR retains data for the last eight days. Any high-load SQL that ran within the retention period of the AWR can be located and tuned using this method. See "Overview of the Automatic Workload Repository" on page 5-8.
- Cursor cache

The third likely source of input is the cursor cache. This source is used for tuning recent SQL statements that are yet to be captured in the AWR. The cursor cache and AWR together provide the capability to identify and tune high-load SQL statements from the current time going as far back as the AWR retention allows, which by default is at least 8 days.

- SQL Tuning Set

  Another possible input source for the SQL Tuning Advisor is the SQL Tuning Set. A SQL Tuning Set (STS) is a database object that stores SQL statements along with their execution context. An STS can include SQL statements that are yet to be deployed, with the goal of measuring their individual performance, or identifying the ones whose performance falls short of expectation. When a set of SQL statements are used as input, a SQL Tuning Set (STS) has to be first constructed and stored. See "SQL Tuning Sets" on page 17-14.

Tuning Options

The SQL Tuning Advisor provides options to manage the scope and duration of a tuning task. The scope of a tuning task can be set to limited or comprehensive.

- If the limited option is chosen, the SQL Tuning Advisor produces recommendations based on statistics checks, access path analysis, and SQL structure analysis. SQL Profile recommendations are not generated.
- If the comprehensive option is selected, the SQL Tuning Advisor carries out all the analysis it performs under limited scope plus SQL Profiling. With the comprehensive option you can also specify a time limit for the tuning task, which by default is 30 minutes.

Advisor Output

After analyzing the SQL statements, the SQL Tuning Advisor provides advice on optimizing the execution plan, the rationale for the proposed optimization, the estimated performance benefit, and the command to implement the advice. You simply have to choose whether or not to accept the recommendations to optimize the SQL statements.

Running the SQL Tuning Advisor

The recommended interface for running the SQL Tuning Advisor is the Oracle Enterprise Manager. Whenever possible, you should run the SQL Tuning Advisor using Oracle Enterprise Manager, as described in the *Oracle Database 2 Day + Performance Tuning Guide*. If Oracle Enterprise Manager is unavailable, you can run the SQL Tuning Advisor using procedures in the `DBMS_SQLTUNE` package. To use the APIs, the user must be granted specific privileges.

> **See Also:** *Oracle Database PL/SQL Packages and Types Reference* for information on the security model for the `DBMS_SQLTUNE` package

Running SQL Tuning Advisor using `DBMS_SQLTUNE` package is a multi-step process:

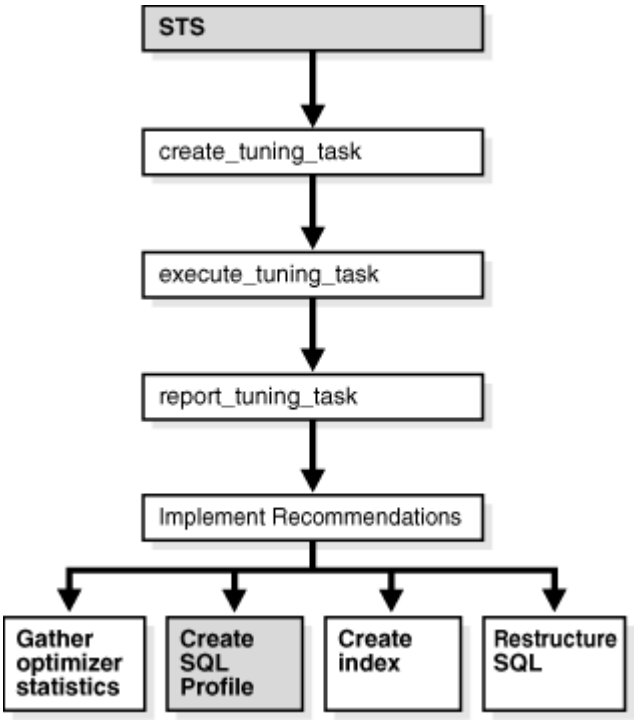
1. Create a SQL Tuning Set (if tuning multiple SQL statements)
2. Create a SQL tuning task
3. Execute a SQL tuning task
4. Display the results of a SQL tuning task

5. Implement recommendations as appropriate

A SQL tuning task can be created for a single SQL statement. For tuning multiple statements, a SQL Tuning Set (STS) has to be first created. An STS is a database object that stores SQL statements along with their execution context. An STS can be created manually using command line APIs or automatically using Oracle Enterprise Manager. See "SQL Tuning Sets" on page 17-14.

Figure 17–2 shows the steps involved when running the SQL Tuning Advisor using the `DBMS_SQLTUNE` package.

Figure 17–2 SQL Tuning Advisor APIs



This section covers the following topics:

- Creating a SQL Tuning Task
- Configuring a SQL Tuning Task
- Executing a SQL Tuning Task
- Checking the Status of a SQL Tuning Task
- Checking the Progress of the SQL Tuning Advisor
- Displaying the Results of a SQL Tuning Task
- Additional Operations on a SQL Tuning Task

> **See Also:** *Oracle Database PL/SQL Packages and Types Reference* for information about the `DBMS_SQLTUNE` package

Creating a SQL Tuning Task

You can create tuning tasks from the text of a single SQL statement, a SQL Tuning Set containing multiple statements, a SQL statement selected by SQL identifier from the

cursor cache, or a SQL statement selected by SQL identifier from the Automatic Workload Repository.

For example, to use the SQL Tuning Advisor to optimize a specified SQL statement text, you need to create a tuning task with the SQL statement passed as a CLOB argument. For the following PL/SQL code, the user HR has been granted the ADVISOR privilege and the function is run as user HR on the employees table in the HR schema.

```
DECLARE
 my_task_name VARCHAR2(30);
 my_sqltext   CLOB;
BEGIN
 my_sqltext := 'SELECT /*+ ORDERED */ * '                      ||
               'FROM employees e, locations l, departments d ' ||
               'WHERE e.department_id = d.department_id AND '  ||
                     'l.location_id = d.location_id AND '      ||
                     'e.employee_id < :bnd';

 my_task_name := DBMS_SQLTUNE.CREATE_TUNING_TASK(
         sql_text    => my_sqltext,
         bind_list   => sql_binds(anydata.ConvertNumber(100)),
         user_name   => 'HR',
         scope       => 'COMPREHENSIVE',
         time_limit  => 60,
         task_name   => 'my_sql_tuning_task',
         description => 'Task to tune a query on a specified employee');
END;
/
```

In this example, 100 is the value for bind variable :bnd passed as function argument of type SQL_BINDS, HR is the user under which the CREATE_TUNING_TASK function analyzes the SQL statement, the scope is set to COMPREHENSIVE which means that the advisor also performs SQL Profiling analysis, and 60 is the maximum time in seconds that the function can run. In addition, values for task name and description are provided.

The CREATE_TUNING_TASK function returns the task name that you have provided or generates a unique task name. You can use the task name to specify this task when using other APIs. To view the task names associated with a specific owner, you can run the following:

```
SELECT task_name FROM DBA_ADVISOR_LOG WHERE owner = 'HR';
```

Configuring a SQL Tuning Task

You can fine tune a SQL tuning task after it has been created by configuring its parameters using the SET_TUNING_TASK_PARAMETER procedure in the DBMS_SQLTUNE package:

```
BEGIN
  DBMS_SQLTUNE.SET_TUNING_TASK_PARAMETER(
    task_name => 'my_sql_tuning_task',
    parameter => 'TIME_LIMIT', value => 300);
END;
/
```

In this example, the maximum time that the SQL tuning task can run is changed to 300 seconds.

Table 17–2 lists the parameters that can be configured using the SET_TUNING_TASK_PARAMETER procedure.

Table 17–2 SET_TUNING_TASK_PARAMETER Procedure Parameters

Parameter	Description
MODE	Specifies the scope of the tuning task: ■ LIMITED takes approximately 1 second to tune each SQL statement but does not recommend a SQL profile ■ COMPREHENSIVE performs a complete analysis and recommends a SQL profile, when appropriate, but may take much longer.
USERNAME	Username under which the SQL statement will be parsed
DAYS_TO_EXPIRE	Number of days before the task is deleted
DEFAULT_EXECUTION_TYPE	Default execution type if not specified by the EXECUTE_TUNING_TASK function when the task is executed
TIME_LIMIT	Time limit (in number of seconds) before the task times out
LOCAL_TIME_LIMIT	Time limit (in number of seconds) for each SQL statement
TEST_EXECUTE	Determines if the SQL Tuning Advisor will test execute the SQL statements to verify the recommendation benefit: ■ FULL - Test executes SQL statements for as much of the local time limit as necessary ■ AUTO - Test executes SQL statements using an automatic time limit ■ OFF - Will not test execute SQL statements
BASIC_FILTER	Basic filter used for SQL tuning set
OBJECT_FILTER	Object filter used for SQL tuning set
PLAN_FILTER	Plan filter used for SQL tuning set
RANK_MEASURE1	First ranking measure used for SQL tuning set
RANK_MEASURE2	Second ranking measure used for SQL tuning set
RANK_MEASURE3	Third ranking measure used for SQL tuning set
RESUME_FILTER	Extra filter used for SQL tuning set (besides BASIC_FILTER)
SQL_LIMIT	Maximum number of SQL statements to tune
SQL_PERCENTAGE	Percentage filter of statements from SQL tuning set

Executing a SQL Tuning Task

After you have created a tuning task, you need to execute the task and start the tuning process. For example:

```
BEGIN
  DBMS_SQLTUNE.EXECUTE_TUNING_TASK( task_name => 'my_sql_tuning_task' );
END;
/
```

Like any other SQL Tuning Advisor task, you can also execute the automatic tuning task SYS_AUTO_SQL_TUNING_TASK using the EXECUTE_TUNING_TASK API. The SQL Tuning Advisor will perform the same analysis and actions as it would when run automatically. You can also pass an execution name to the API to name the new execution.

Checking the Status of a SQL Tuning Task

You can check the status of the task by reviewing the information in the USER_ADVISOR_TASKS view or check execution progress of the task in the V$SESSION_LONGOPS view. For example:

```
SELECT status FROM USER_ADVISOR_TASKS WHERE task_name = 'my_sql_tuning_task';
```

Checking the Progress of the SQL Tuning Advisor

You can check the execution progress of the SQL Tuning Advisor in the V$ADVISOR_PROGRESS view. For example:

```
SELECT sofar, totalwork FROM V$ADVISOR_PROGRESS WHERE user_name = 'HR' AND task_
name = 'my_sql_tuning_task';
```

> **See Also:** *Oracle Database Reference* for information on the V$ADVISOR_PROGRESS view

Displaying the Results of a SQL Tuning Task

After a task has been executed, you display a report of the results with the REPORT_TUNING_TASK function. For example:

```
SET LONG 1000
SET LONGCHUNKSIZE 1000
SET LINESIZE 100
SELECT DBMS_SQLTUNE.REPORT_TUNING_TASK( 'my_sql_tuning_task')
  FROM DUAL;
```

The report contains all the findings and recommendations of the SQL Tuning Advisor. For each proposed recommendation, the rationale and benefit is provided along with the SQL commands needed to implement the recommendation.

Additional information about tuning tasks and results can be found in DBA views. See "SQL Tuning Information Views" on page 17-21.

Additional Operations on a SQL Tuning Task

You can use the following APIs for managing SQL tuning tasks:

- INTERRUPT_TUNING_TASK to interrupt a task while executing, causing a normal exit with intermediate results
- RESUME_TUNING_TASK to resume a previously interrupted task
- CANCEL_TUNING_TASK to cancel a task while executing, removing all results from the task
- RESET_TUNING_TASK to reset a task while executing, removing all results from the task and returning the task to its initial state
- DROP_TUNING_TASK to drop a task, removing all results associated with the task

SQL Tuning Sets

A SQL Tuning Set (STS) is a database object that includes one or more SQL statements along with their execution statistics and execution context, and could include a user priority ranking. The SQL statements can be loaded into a SQL Tuning Set from different SQL sources, such as the Automatic Workload Repository, the cursor cache, or custom SQL provided by the user. An STS includes:

- A set of SQL statements
- Associated execution context, such as user schema, application module name and action, list of bind values, and the cursor compilation environment
- Associated basic execution statistics, such as elapsed time, CPU time, buffer gets, disk reads, rows processed, cursor fetches, the number of executions, the number of complete executions, optimizer cost, and the command type
- Associated execution plans and row source statistics for each SQL statement (optional)

SQL statements can be filtered using the application module name and action, or any of the execution statistics. In addition, the SQL statements can be ranked based on any combination of execution statistics.

A SQL Tuning Set can be used as input to the SQL Tuning Advisor, which performs automatic tuning of the SQL statements based on other input parameters specified by the user. SQL Tuning Sets are transportable across databases and can be exported from one system to another, allowing for the transfer of SQL workloads between databases for remote performance diagnostics and tuning. When poorly performing SQL statements are encountered on a production system, it may not be desirable for developers to perform their investigation and tuning activities on the production system directly. This feature allows the DBA to transport the problematic SQL statements to a test system where the developers can safely analyze and tune them. To transport SQL Tuning Sets, use the `DBMS_SQLTUNE` package procedures.

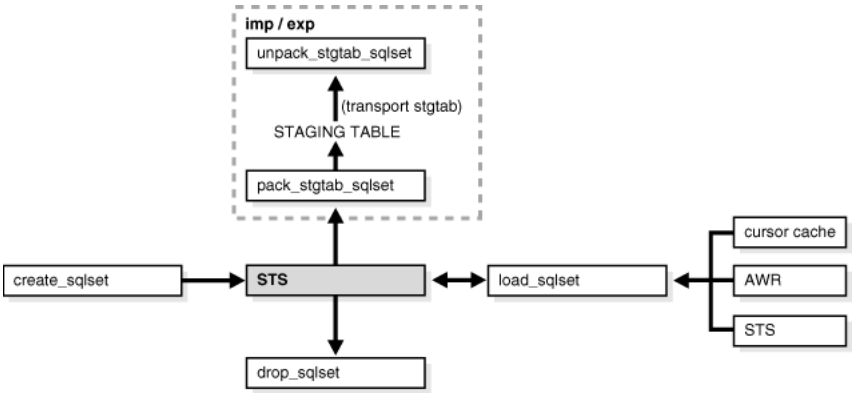
The recommended interface for managing SQL tuning sets is the Oracle Enterprise Manager. Whenever possible, you should manage SQL tuning sets using Oracle Enterprise Manager, as described in the *Oracle Database 2 Day + Performance Tuning Guide*. If Oracle Enterprise Manager is unavailable, you can manage SQL tuning sets using the `DBMS_SQLTUNE` package procedures. Typically you would use the STS operations in the following sequence:

- Create a new STS
- Load the STS
- Select the STS to review the contents
- Update the STS if necessary
- Create a tuning task with the STS as input
- Transporting the STS to another system if necessary
- Drop the STS when finished

To use the APIs, you need the `ADMINISTER SQL TUNING SET` system privilege to manage SQL Tuning Sets that you own, or the `ADMINISTER ANY SQL TUNING SET` system privilege to manage any SQL Tuning Sets.

Figure 17–3 shows the steps involved when using SQL Tuning Sets APIs.

Figure 17–3 SQL Tuning Sets APIs



This section covers the following topics:

- Creating a SQL Tuning Set
- Loading a SQL Tuning Set
- Displaying the Contents of a SQL Tuning Set
- Modifying a SQL Tuning Set
- Transporting a SQL Tuning Set
- Dropping a SQL Tuning Set
- Additional Operations on SQL Tuning Sets

> **See Also:** *Oracle Database PL/SQL Packages and Types Reference* for information about the `DBMS_SQLTUNE` package

Creating a SQL Tuning Set

The `CREATE_SQLSET` procedure is used to create an empty STS object in the database. For example, the following procedure creates an STS object that could be used to tune I/O intensive SQL statements during a specific period of time:

```
BEGIN
  DBMS_SQLTUNE.CREATE_SQLSET(
    sqlset_name => 'my_sql_tuning_set',
    description  => 'I/O intensive workload');
END;
/
```

where `my_sql_tuning_set` is the name of the STS in the database and `'I/O intensive workload'` is the description assigned to the STS.

Loading a SQL Tuning Set

The `LOAD_SQLSET` procedure populates the STS with selected SQL statements. The standard sources for populating an STS are the workload repository, another STS, or the cursor cache. For both the workload repository and STS, there are predefined table functions that can be used to select columns from the source to populate a new STS.

In the following example, procedure calls are used to load `my_sql_tuning_set` from an AWR baseline called `peak baseline`. The data has been filtered to select only the top 30 SQL statements ordered by elapsed time. First a ref cursor is opened to

select from the specified baseline. Next the statements and their statistics are loaded from the baseline into the STS.

```
DECLARE
 baseline_cursor DBMS_SQLTUNE.SQLSET_CURSOR;
BEGIN
 OPEN baseline_cursor FOR
    SELECT VALUE(p)
    FROM TABLE (DBMS_SQLTUNE.SELECT_WORKLOAD_REPOSITORY(
                  'peak baseline',
                   NULL, NULL,
                   'elapsed_time',
                   NULL, NULL, NULL,
                   30)) p;

    DBMS_SQLTUNE.LOAD_SQLSET(
             sqlset_name     => 'my_sql_tuning_set',
             populate_cursor => baseline_cursor);
END;
/
```

Displaying the Contents of a SQL Tuning Set

The SELECT_SQLSET table function reads the contents of the STS. After an STS has been created and populated, you can browse the SQL in the STS using different filtering criteria. The SELECT_SQLSET procedure is provided for this purpose.

In the following example, the SQL statements in the STS are displayed for statements with a disk-reads to buffer-gets ratio greater than or equal to 75%.

```
SELECT * FROM TABLE(DBMS_SQLTUNE.SELECT_SQLSET(
   'my_sql_tuning_set',
   '(disk_reads/buffer_gets) >= 0.75'));
```

Additional details of the SQL Tuning Sets that have been created and loaded can also be displayed with DBA views, such as DBA_SQLSET, DBA_SQLSET_STATEMENTS, and DBA_SQLSET_BINDS.

Modifying a SQL Tuning Set

SQL statements can be updated and deleted from a SQL Tuning Set based on a search condition. In the following example, the DELETE_SQLSET procedure deletes SQL statements from my_sql_tuning_set that have been executed less than fifty times.

```
BEGIN
  DBMS_SQLTUNE.DELETE_SQLSET(
      sqlset_name  => 'my_sql_tuning_set',
      basic_filter => 'executions < 50');
END;
/
```

Transporting a SQL Tuning Set

SQL Tuning Sets can be transported to another system by first exporting the STS from one system to a staging table, then importing the STS from the staging table into another system.

To transport a SQL Tuning Set:

1. Use the CREATE_STGTAB_SQLSET procedure to create a staging table where the SQL Tuning Sets will be exported.

The following example shows how to create a staging table named `staging_table`. Table names are case-sensitive.

```
BEGIN
  DBMS_SQLTUNE.CREATE_STGTAB_SQLSET( table_name => 'staging_table' );
END;
/
```

2. Use the `PACK_STGTAB_SQLSET` procedure to export SQL Tuning Sets into the staging table.

   The following example shows how to export a SQL Tuning Set named `my_sts` to the staging table.

```
BEGIN
  DBMS_SQLTUNE.PACK_STGTAB_SQLSET(
      sqlset_name  => 'my_sts',
      staging_table_name => 'staging_table');
END;
/
```

3. Move the staging table to the system where the SQL Tuning Sets will be imported using the mechanism of choice (such as datapump or database link).

4. On the system where the SQL Tuning Sets will be imported, use the `UNPACK_STGTAB_SQLSET` procedure to import SQL Tuning Sets from the staging table.

   The following example shows how to import SQL Tuning Sets contained in the staging table.

```
BEGIN
  DBMS_SQLTUNE.UNPACK_STGTAB_SQLSET(
      sqlset_name  => '%',
      replace  => TRUE,
      staging_table_name => 'staging_table');
END;
/
```

Dropping a SQL Tuning Set

The `DROP_SQLSET` procedure is used to drop an STS that is no longer needed. For example:

```
BEGIN
  DBMS_SQLTUNE.DROP_SQLSET( sqlset_name => 'my_sql_tuning_set' );
END;
/
```

Additional Operations on SQL Tuning Sets

You can use the following APIs to manage an STS:

- Updating the attributes of SQL statements in an STS

  The `UPDATE_SQLSET` procedure updates the attributes of SQL statements (such as `PRIORITY` or `OTHER`) in an existing STS identified by STS name and SQL identifier.

- Capturing the full system workload

  The `CAPTURE_CURSOR_CACHE_SQLSET` function enables the capture of the full system workload by repeatedly polling the cursor cache over a specified interval. This function is a lot more efficient than repeatedly using the `SELECT_CURSOR_CACHE` and `LOAD_SQLSET` procedures to capture the cursor cache over an extended period of time. This function effectively captures the entire workload, as

opposed to the AWR—which only captures the workload of high-load SQL statements—or the LOAD_SQLSET procedure, which accesses the data source only once.

- Adding and removing a reference to an STS

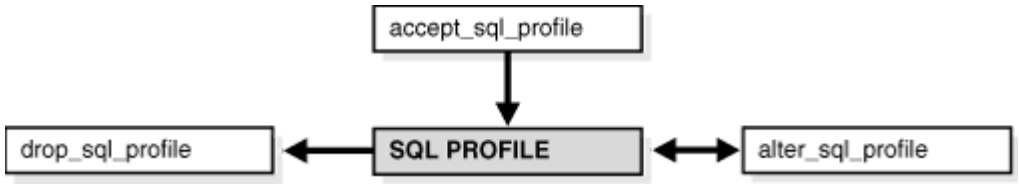
  The ADD_SQLSET_REFERENCE function adds a new reference to an existing STS to indicate its use by a client. The function returns the identifier of the added reference. The REMOVE_SQLSET_REFERENCE procedure is used to deactivate an STS to indicate it is no longer used by the client.

SQL Profiles

While SQL Profiles are usually handled by Oracle Enterprise Manager as part of the Automatic SQL tuning process, SQL Profiles can be managed through the DBMS_SQLTUNE package. To use the SQL Profiles APIs, you need the ADMINISTER SQL MANAGEMENT OBJECT privilege.

Figure 17–4 shows the steps involved when using SQL Profiles APIs.

Figure 17–4 SQL Profiles APIs



This section covers the following topics:

- Accepting a SQL Profile
- Altering a SQL Profile
- Dropping a SQL Profile

> **See Also:** *Oracle Database PL/SQL Packages and Types Reference* for information about the DBMS_SQLTUNE package

Accepting a SQL Profile

When the SQL Tuning Advisor recommends that a SQL Profile be used, you should accept the SQL Profile that is recommended. In cases where the SQL Tuning Advisor recommends that an index and a SQL Profile be used, both should be used. You can use the DBMS_SQLTUNE.ACCEPT_SQL_PROFILE procedure to accept a SQL Profile recommended by the SQL Tuning Advisor. This creates and stores a SQL Profile in the database. For example:

```
DECLARE
 my_sqlprofile_name VARCHAR2(30);
BEGIN
 my_sqlprofile_name := DBMS_SQLTUNE.ACCEPT_SQL_PROFILE (
    task_name    => 'my_sql_tuning_task',
    name         => 'my_sql_profile',
    force_match  => TRUE);
END;
```

In this example, my_sql_tuning_task is the name of the SQL tuning task and my_sql_profile is the name of the SQL Profile that you want to accept.

Typically, an accepted SQL Profile is associated with the SQL statement through a special SQL signature that is generated using a hash function. This hash function normalizes the SQL statement for case (changes the entire SQL statement to upper case) and white spaces (removes all extra whites spaces) before generating the signature. The same SQL Profile thus will work for all SQL statements that are essentially the same, where the only difference is in case usage and white spaces. However, by setting `force_match` to true, the SQL Profile will additionally target all SQL statements that have the same text after normalizing literal values to bind variables. This may be useful for applications that use literal values rather than bind variables, since this will allow SQL with text differing only in its literal values to share a SQL Profile. If both literal values and bind variables are used in the SQL text, or if this parameter is set to false (the default value), literal values will not be normalized.

If SQL plan management is used and there is already an existing plan baseline for the SQL statement, a new plan baseline will be added when a SQL profile is created. If SQL plan management is not used, a new plan baseline will not be added when a SQL profile is created. There is no strict relationship between the SQL profile and the plan baseline. When hard parsing a SQL statement, the optimizer will use the SQL profile to select the best plan baseline from the ones available. In different conditions, the SQL profile may cause the optimizer to select different plan baselines. For information about SQL plan management, see Chapter 15, "Using SQL Plan Management".

You can view information about a SQL Profile in the `DBA_SQL_PROFILES` view.

Altering a SQL Profile

You can alter the `STATUS`, `NAME`, `DESCRIPTION`, and `CATEGORY` attributes of an existing SQL Profile with the `ALTER_SQL_PROFILE` procedure. For example:

```
BEGIN
  DBMS_SQLTUNE.ALTER_SQL_PROFILE(
     name            => 'my_sql_profile',
     attribute_name  => 'STATUS',
     value           => 'DISABLED');
END;
/
```

In this example, `my_sql_profile` is the name of the SQL Profile that you want to alter. The status attribute is changed to disabled, which means the SQL Profile is not used during SQL compilation.

Dropping a SQL Profile

You can drop a SQL Profile with the `DROP_SQL_PROFILE` procedure. For example:

```
BEGIN
  DBMS_SQLTUNE.DROP_SQL_PROFILE(name => 'my_sql_profile');
END;
/
```

In this example, `my_sql_profile` is the name of the SQL Profile you want to drop. You can also specify whether to ignore errors raised if the name does not exist. For this example, the default value of `FALSE` is accepted.

SQL Tuning Information Views

This section summarizes the views that you can display to review information that has been gathered for tuning the SQL statements. You need DBA privileges to access these views.

- Advisor information views, such as DBA_ADVISOR_TASKS, DBA_ADVISOR_EXECUTIONS, DBA_ADVISOR_FINDINGS, DBA_ADVISOR_RECOMMENDATIONS, and DBA_ADVISOR_RATIONALE views.
- SQL tuning information views, such as DBA_SQLTUNE_STATISTICS, DBA_SQLTUNE_BINDS, and DBA_SQLTUNE_PLANS views.
- SQL Tuning Set views, such as DBA_SQLSET, DBA_SQLSET_BINDS, DBA_SQLSET_STATEMENTS, and DBA_SQLSET_REFERENCES views.
- Information on captured execution plans for statements in SQL Tuning Sets are displayed in the DBA_SQLSET_PLANS and USER_SQLSET_PLANS views.
- SQL Profile information is displayed in the DBA_SQL_PROFILES view.

  The TYPE parameter shows if the SQL profile was created manually by the SQL Tuning Advisor (if TYPE = MANUAL) or automatically by automatic SQL tuning (if TYPE = AUTO).
- Advisor execution progress information is displayed in the V$ADVISOR_PROGRESS view.
- Dynamic views containing information relevant to the SQL tuning, such as VSQL, VSQLAREA, V$SQLSTATS, and V$SQL_BINDS views.

> **See Also:** *Oracle Database Reference* for information on static data dictionary and dynamic views

18

SQL Access Advisor

This chapter illustrates how to use the SQL Access Advisor, which is a tuning tool that provides advice on improving the performance of a database through partitioning, materialized views, indexes, and materialized view logs. The chapter contains the following sections:

Overview of the SQL Access Advisor

Materialized views, partitions, and indexes are essential when tuning a database to achieve optimum performance for complex, data-intensive queries. The SQL Access Advisor helps you achieve your performance goals by recommending the proper set of materialized views, materialized view logs, partitions, and indexes for a given workload. Understanding and using these structures is essential when optimizing SQL as they can result in significant performance improvements in data retrieval. The advantages, however, do not come without a cost. Creation and maintenance of these objects can be time consuming, and space requirements can be significant. In particular, partitioning of an unpartitioned base table is a complex operation that must be planned carefully.

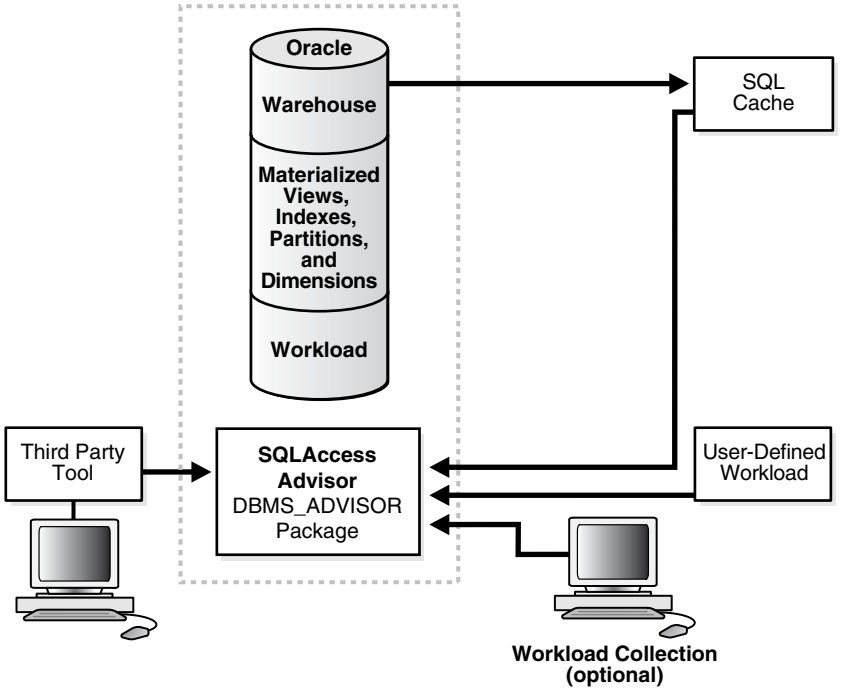
The SQL Access Advisor index recommendations include bitmap, function-based, and B-tree indexes. A bitmap index offers a reduced response time for many types of ad hoc queries and reduced storage requirements compared to other indexing techniques. B-tree indexes are most commonly used in a data warehouse to index unique or near-unique keys.

The SQL Access Advisor, using the `TUNE_MVIEW` procedure, also recommends how to optimize materialized views so that they can be fast refreshable and take advantage of general query rewrite.

In addition, the SQL Access Advisor has the ability to recommend partitioning on an existing unpartitioned base table in order to improve performance. Furthermore, it may recommend new indexes and materialized views that are themselves partitioned. While creating new partitioned indexes and materialized view is no different from the unpartitioned case, partitioning existing base tables should be executed with care. This is especially true if indexes, views, constraints, or triggers are already defined on the table. See "Special Considerations when Script Includes Partitioning Recommendations" on page 18-20 for a list of issues involving base table partitioning and the `DBMS_REDEFINITION` package for performing this task online.

The SQL Access Advisor can be run from Oracle Enterprise Manager (accessible from the Advisor Central page) using the SQL Access Advisor Wizard or by invoking the DBMS_ADVISOR package. The DBMS_ADVISOR package consists of a collection of analysis and advisory functions and procedures callable from any PL/SQL program. Figure 18–1 illustrates how the SQL Access Advisor recommends materialized views for a given workload obtained from a user-defined table or the SQL cache. If a workload is not provided, it can generate and use a hypothetical workload also, provided the user schema contains dimensions defined by the CREATE DIMENSION keyword.

Figure 18–1 Materialized Views and the SQL Access Advisor



Using the SQL Access Advisor in Enterprise Manager or API, you can do the following:

- Recommend materialized views and indexes based on collected, user-supplied, or hypothetical workload information.
- Recommend partitioning of tables, indexes, partitions, and materialized views.
- Mark, update, and remove recommendations.

In addition, you can use the SQL Access Advisor API to do the following:

- Perform a quick tune using a single SQL statement.
- Show how to make a materialized view fast refreshable.
- Show how to change a materialized view so that general query rewrite is possible.

In order to make recommendations, the SQL Access Advisor relies on structural statistics about table and index cardinalities of dimension level columns, JOIN KEY columns, and fact table key columns. You can gather either exact or estimated statistics with the DBMS_STATS package. Because gathering statistics is time-consuming and full statistical accuracy is not required, it is generally preferable to estimate statistics. Without gathering statistics on a given table, any queries referencing that table will be marked as invalid in the workload, resulting in no recommendations being made for

those queries. It is also recommended that all existing indexes and materialized views have been analyzed. See *Oracle Database PL/SQL Packages and Types Reference* for more information regarding the DBMS_STATS package.

Overview of Using the SQL Access Advisor

One of the easiest ways to use the SQL Access Advisor is to invoke its wizard, which is available in Oracle Enterprise Manager from the Advisor Central page. If you prefer to use SQL Access Advisor through the DBMS_ADVISOR package, this section describes the basic components and the sequence in which the various procedures must be called.

This section describes the four steps in generating a set of recommendations:

- Create a task
- Define the workload
- Generate the recommendations
- View and implement the recommendations

Step 1 Create a task

Before any recommendations can be made, a task must be created. The task is important because it is where all information relating to the recommendation process resides, including the results of the recommendation process. If you use the wizard in Oracle Enterprise Manager or the DBMS_ADVISOR.QUICK_TUNE procedure, the task is created automatically for you. In all other cases, you must create a task using the DBMS_ADVISOR.CREATE_TASK procedure.

You can control what a task does by defining parameters for that task using the DBMS_ADVISOR.SET_TASK_PARAMETER procedure.

See "Creating Tasks" on page 18-7 for more information about creating tasks.

Step 2 Define the workload

The workload is one of the primary inputs for the SQL Access Advisor, and it consists of one or more SQL statements, plus various statistics and attributes that fully describe each statement. If the workload contains all SQL statements from a target business application, the workload is considered a full workload; if the workload contains a subset of SQL statements, it is known as a partial workload. The difference between a full and a partial workload is that in the former case, the SQL Access Advisor may recommend dropping certain existing materialized views and indexes if it finds that they are not being used.

Typically, the SQL Access Advisor uses the workload as the basis for all analysis activities. Although the workload may contain a wide variety of statements, it carefully ranks the entries according to a specific statistic, business importance, or a combination of statistics and business importance. This ranking is critical in that it enables the SQL Access Advisor to process the most important SQL statements ahead of those with less business impact.

For a collection of data to be considered a valid workload, the SQL Access Advisor may require particular attributes to be present. Although analysis can be performed if some of the items are missing, the quality of the recommendations may be greatly diminished. For example, the SQL Access Advisor requires a workload to contain a SQL query and the user who executed the query. All other attributes are optional; however, if the workload also contained I/O and CPU information, then the SQL Access Advisor may be able to better evaluate the current efficiency of the statement.

The workload is stored as a SQL Tuning Set object, which is accessed using the DBMS_SQLTUNE package, and can easily be shared among many Advisor tasks. Because the workload is independent, it must be linked to a task using the DBMS_ADVISOR.ADD_STS_REF procedure. Once this link has been established, the workload cannot be deleted or modified until all Advisor tasks have removed their dependency on the workload. A workload reference will be removed when a parent Advisor task is deleted or when the workload reference is manually removed from the Advisor task by the user using the DBMS_ADVISOR.DELETE_STS_REF procedure.

You cannot use the SQL Access Advisor without a workload, however, it is possible to create a hypothetical workload from a schema by analyzing dimensions and constraints. For best results, an actual workload should be provided in the form of a SQL Tuning Set.

The DBMS_SQLTUNE package provides several helper functions that can create SQL Tuning Sets from common workload sources, such as the SQL cache, a user-defined workload stored in a table and a hypothetical workload.

At the time the recommendations are generated, a filter can be applied to the workload to restrict what is analyzed. This provides the ability to generate different sets of recommendations based on different workload scenarios.

The recommendation process and customization of the workload are controlled by SQL Access Advisor parameters. These parameters control various aspects of the recommendation process, such as the type of recommendation that is required and the naming conventions for what it recommends.

To set these parameters, use the SET_TASK_PARAMETER procedure. Parameters are persistent in that they remain set for the lifespan of the task. When a parameter value is set using the SET_TASK_PARAMETER procedure, it does not change until you make another call to SET_TASK_PARAMETER.

Step 3 Generate the recommendations

Once a task exists and a workload is linked to the task and the appropriate parameters are set, you can generate recommendations using the DBMS_ADVISOR.EXECUTE_TASK procedure. These recommendations are stored in the SQL Access Advisor Repository.

The recommendation process generates a number of recommendations and each recommendation will be comprised of one or more actions. For example, a recommendation could be to create a number of materialized view logs, create a materialized view, and then analyze it to gather statistical information.

A task recommendation can range from a simple suggestion to a complex solution that requires partitioning a set of existing base tables and implementing a set of database objects such as indexes, materialized views, and materialized view logs. When an Advisor task is executed, the SQL Access Advisor carefully analyzes collected data and user-adjusted task parameters. It then forms a structured recommendation that can be viewed and implemented by the user.

See "Generating Recommendations" on page 18-12 for more information about generating recommendations.

Step 4 View and implement the recommendations

There are two ways to view the recommendations from the SQL Access Advisor: using the catalog views or by generating a script using the DBMS_ADVISOR.GET_TASK_SCRIPT procedure. In Enterprise Manager, the recommendations may be displayed once the SQL Access Advisor process has completed. See "Viewing Recommendations" on page 18-12 for a description of using the catalog views to view the

recommendations. See "Generating SQL Scripts" on page 18-19 to see how to create a script.

Not all recommendations have to be accepted and you can mark the ones that should be included in the recommendation script. However, when base table partitioning is recommended, some recommendations depend on others (for example, you cannot implement a local index if you do not also implement the partitioning recommendation on the index base table).

The final step is then implementing the recommendations and verifying that query performance has improved.

SQL Access Advisor Repository

All the information needed and generated by the SQL Access Advisor is held in the Advisor repository, which is a part of the database dictionary. The benefits of using the repository are that it:

- Collects a complete workload for the SQL Access Advisor.
- Supports historical data.
- Is managed by the server.

Using the SQL Access Advisor

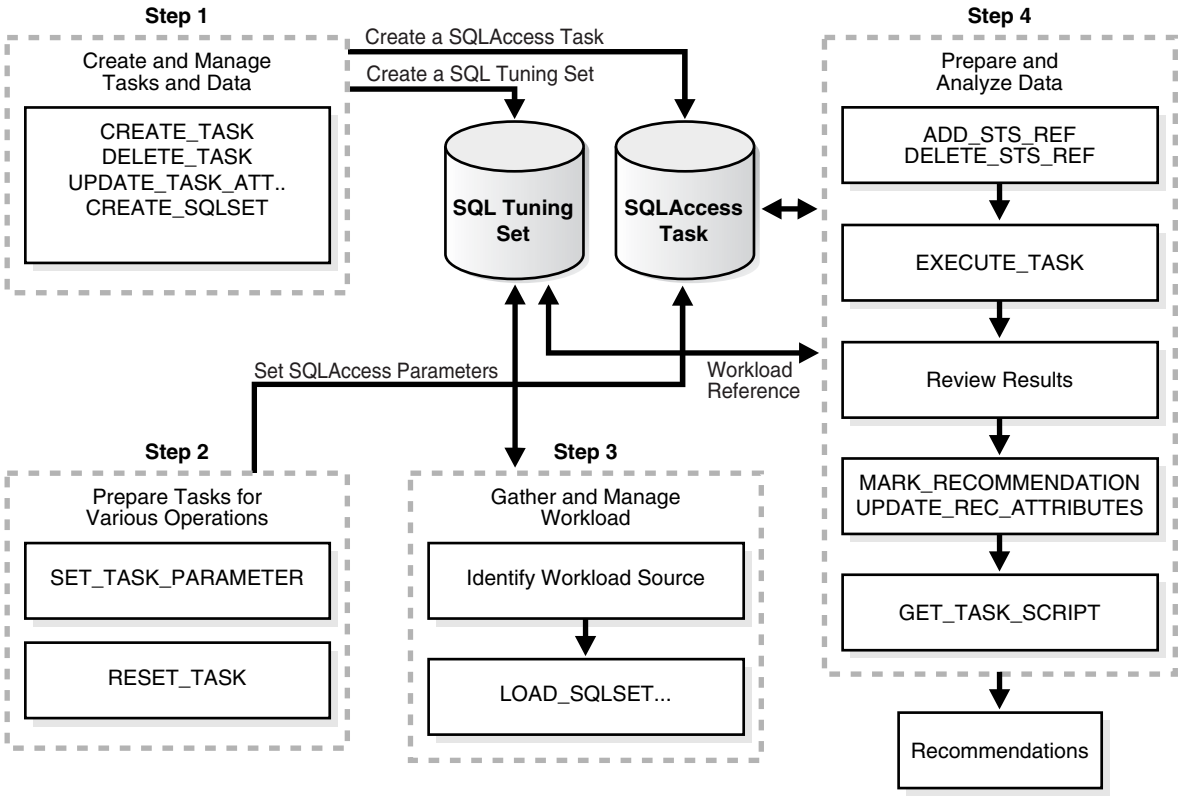
This section discusses general information about, as well as the steps needed to use, the SQL Access Advisor, and includes:

- Steps for Using the SQL Access Advisor
- Privileges Needed to Use the SQL Access Advisor
- Setting Up Tasks and Templates
- SQL Access Advisor Workloads
- Working with Recommendations
- Performing a Quick Tune
- Managing Tasks
- Using SQL Access Advisor Constants

Steps for Using the SQL Access Advisor

Figure 18–2 illustrates the steps in using the SQL Access Advisor as well as an overview of all of the parameters in the SQL Access Advisor and when it is appropriate to use them.

Figure 18–2 SQL Access Advisor Flowchart



Privileges Needed to Use the SQL Access Advisor

You need to have the `ADVISOR` privilege to manage or use the SQL Access Advisor. When processing a workload, the SQL Access Advisor attempts to validate each statement in order to identify table and column references. Validation is achieved by processing each statement as if it is being executed by the statement's original user. If that user does not have `SELECT` privileges to a particular table, the SQL Access Advisor bypasses the statement referencing the table. This can cause many statements to be excluded from analysis. If the SQL Access Advisor excludes all statements in a workload, the workload is invalid and the SQL Access Advisor returns the following message:

```
QSM-00774, there are no SQL statements to process for task TASK_NAME
```

To avoid missing critical workload queries, the current database user must have `SELECT` privileges on the tables targeted for materialized view analysis. For those tables, these `SELECT` privileges cannot be obtained through a role.

Additionally, you must have the `ADMINISTER SQL TUNING SET` privilege in order to create and manage workloads in SQL Tuning Set objects. If you want to run the Advisor on SQL Tuning Sets owned by other users, you must have the `ADMINISTER ANY SQL TUNING SET` privilege.

Setting Up Tasks and Templates

This section discusses the following aspects of setting up tasks and templates:

- Creating Tasks
- Using Templates

Creating Tasks

An Advisor task is where you define what it is you want to analyze and where the results of this analysis should be placed. A user can create any number of tasks, each with its own specialization. All are based on the same Advisor task model and share the same repository.

You create a task using the CREATE_TASK procedure. The syntax is as follows:

```
DBMS_ADVISOR.CREATE_TASK (
   advisor_name          IN VARCHAR2,
   task_id               OUT NUMBER,
   task_name             IN OUT VARCHAR2,
   task_desc             IN VARCHAR2 := NULL,
   template              IN VARCHAR2 := NULL,
   is_template           IN VARCHAR2 := 'FALSE',
   how_created           IN VARCHAR2 := NULL);
```

The following illustrates an example of using this procedure:

```
VARIABLE task_id NUMBER;
VARIABLE task_name VARCHAR2(255);
EXECUTE :task_name := 'MYTASK';
EXECUTE DBMS_ADVISOR.CREATE_TASK ('SQL Access Advisor', :task_id, :task_name);
```

See *Oracle Database PL/SQL Packages and Types Reference* for more information regarding the CREATE_TASK procedure and its parameters.

Using Templates

When an ideal configuration for a task or workload has been identified, this configuration can be saved as a template upon which future tasks and workloads can be based.

This enables you to set up any number of tasks or workloads that can be used as intelligent starting points or templates for future task creation. By setting up a template, you can save time when performing tuning analysis. It also enables you to custom fit a tuning analysis to the business operation.

To create a task from a template, you specify the template to be used when a new task is created. At that time, the SQL Access Advisor copies the data and parameter settings from the template into the newly created task. You can also set an existing task to be a template by setting the template attribute when creating the task or later using the UPDATE_TASK_ATTRIBUTE procedure.

To use a task as a template, you tell the SQL Access Advisor to use a task when a new task is created. At that time, the SQL Access Advisor copies the task template's data and parameter settings into the newly created task. You can also set an existing task to be a template by setting the template attribute. This can be done at the command line or in Enterprise Manager.

Creating Templates

You can create a template as in the following example.

1. Create a template called MY_TEMPLATE.

```
VARIABLE template_id NUMBER;
VARIABLE template_name VARCHAR2(255);
EXECUTE :template_name := 'MY_TEMPLATE';
```

```
EXECUTE DBMS_ADVISOR.CREATE_TASK('SQL Access Advisor',:template_id, -
                                 :template_name, is_template => 'TRUE');
```

2. Set template parameters. For example, the following sets the naming conventions for recommended indexes and materialized views and the default tablespaces:

```
-- set naming conventions for recommended indexes/mvs
EXECUTE DBMS_ADVISOR.SET_TASK_PARAMETER ( -
   :template_name, 'INDEX_NAME_TEMPLATE', 'SH_IDX$$_<SEQ>');

EXECUTE DBMS_ADVISOR.SET_TASK_PARAMETER ( -
   :template_name, 'MVIEW_NAME_TEMPLATE', 'SH_MV$$_<SEQ>');

-- set default tablespace for recommended indexes/mvs
EXECUTE DBMS_ADVISOR.SET_TASK_PARAMETER ( -
   :template_name, 'DEF_INDEX_TABLESPACE', 'SH_INDEXES');

EXECUTE DBMS_ADVISOR.SET_TASK_PARAMETER ( -
   :template_name, 'DEF_MVIEW_TABLESPACE', 'SH_MVIEWS');
```

3. This template can now be used as a starting point to create a task as follows:

```
VARIABLE task_id NUMBER;
VARIABLE task_name VARCHAR2(255);
EXECUTE :task_name := 'MYTASK';
EXECUTE DBMS_ADVISOR.CREATE_TASK('SQL Access Advisor', :task_id, -
                                 :task_name, template=>'MY_TEMPLATE');
```

   The following example uses a pre-defined template `SQLACCESS_WAREHOUSE`. See Table 18–3 for more information.

```
EXECUTE DBMS_ADVISOR.CREATE_TASK('SQL Access Advisor', -
   :task_id, :task_name,  template=>'SQLACCESS_WAREHOUSE');
```

SQL Access Advisor Workloads

The SQL Access Advisor supports different types of workloads, and this section discusses the following aspects of managing workloads:

- SQL Tuning Set Workloads
- Using SQL Tuning Sets
- Linking Tasks and Workloads

SQL Tuning Set Workloads

The input workload source for the SQL Access Advisor is the SQL Tuning Set. An important benefit of using a SQL Tuning Set is that because it is stored as a separate entity, it can easily be shared among many Advisor tasks. Once a SQL Tuning Set object has been referenced by an Advisor task, it cannot be deleted or modified until all Advisor tasks have removed their dependency on the data. A workload reference will be removed when a parent Advisor task is deleted or when the workload reference is manually removed from the Advisor task by the user.

The SQL Access Advisor performs best when a workload based on actual usage is available. You can store multiple workloads in the form of SQL Tuning Sets, so that the different uses of a real-world data warehousing or transaction-processing environment can be viewed over a long period of time and across the life cycle of database instance startup and shutdown.

Using SQL Tuning Sets

The SQL Tuning Set workload is implemented using the DBMS_SQLTUNE package. See *Oracle Database PL/SQL Packages and Types Reference* for a description on creating and managing SQL Tuning Sets.

To transition existing SQL Workload objects to a SQL Tuning Set, the DBMS_ADVISOR package provides a procedure to copy SQL Workload data to a user-designated SQL Tuning Set. Note that, to use this procedure, the user must have the required SQL Tuning Set privileges as well as the required ADVISOR privilege.

The syntax is as follows:

```
DBMS_ADVISOR.COPY_SQLWKLD_TO_STS (
   workload_name        IN VARCHAR2,
   sts_name             IN VARCHAR2,
   import_mode          IN VARCHAR2 := 'NEW');
```

The following example illustrates its usage:

```
EXECUTE DBMS_ADVISOR.COPY_SQLWKLD_TO_STS('MYWORKLOAD','MYSTS','NEW');
```

See *Oracle Database PL/SQL Packages and Types Reference* for more information regarding the COPY_SQLWKLD_TO_STS procedure and its parameters.

Linking Tasks and Workloads

Before the recommendation process can begin, the task must be linked to a SQL Tuning Set. You achieve this by using the ADD_STS_REF procedure and using their respective names to link the task and a Tuning Set. This procedure establishes a link between the Advisor task and a Tuning Set. And, once a connection has been defined, the SQL Tuning Set is protected from removal or update. The syntax is as follows:

```
DBMS_ADVISOR.ADD_STS_REF (task_name IN VARCHAR2,

sts_owner IN VARCHAR2,
sts_name  IN VARCHAR2);
```

The sts_owner parameter may be null, in which case the STS is assumed to be owned by the current user.

The following example links the MYTASK task created to the current user's MYWORKLOAD SQL Tuning Set:

```
EXECUTE DBMS_ADVISOR.ADD_STS_REF('MYTASK', null, 'MYWORKLOAD');
```

See *Oracle Database PL/SQL Packages and Types Reference* for more information regarding the ADD_STS_REF procedure and its parameters.

Removing a Link Between a SQL Tuning Set Workload and a Task Before a task or a SQL Tuning Set workload can be deleted, if it is linked to a workload or task respectively, then the link between the task and the workload must be removed using the DELETE_STS_REF procedure. The following example deletes the link between task MYTASK and the current user's SQL Tuning Set MYWORKLOAD:

```
EXECUTE DBMS_ADVISOR.DELETE_STS_REF('MYTASK', null, 'MYWORKLOAD');
```

Working with Recommendations

This section discusses the following aspects of working with recommendations:

- Recommendations and Actions

Recommendations and Actions

The advisor will make a number of recommendations, each of which contain one or multiple individual actions. In general, each recommendation provides a benefit for one query or a set of queries. All individual actions in a recommendation must be implemented together to achieve the full benefit. Actions can be shared among recommendations. For example, a `CREATE INDEX` statement could provide a benefit for a number of queries, but some of those queries might benefit from an additional `CREATE MATERIALIZED VIEW` statement. In that case, the advisor would generate two recommendations: one for the set of queries that require only the index, and another one for the set of queries that require both the index and the materialized view to run optimally.

There is one special type of recommendation: the partition recommendation. When the Advisor decides that one or multiple base tables should be partitioned to improve workload performance, it will collect all individual partition actions into a single recommendation. In that case, note that some or all of the remaining recommendations might depend on the partitioning recommendation, because index and materialized view advice cannot be seen in isolation of the underlying tables' partitioning schemes.

Recommendation Options

Before recommendations can be generated, the parameters for the task must first be defined using the `SET_TASK_PARAMETER` procedure. If parameters are not defined, then the defaults are used.

You can set task parameters by using the `SET_TASK_PARAMETER` procedure. The syntax is as follows.

```
DBMS_ADVISOR.SET_TASK_PARAMETER (
   task_name           IN VARCHAR2,
   parameter           IN VARCHAR2,
   value               IN [VARCHAR2 | NUMBER]);
```

There are many task parameters and, to help identify the relevant ones, they have been grouped into categories in Table 18–1. Note that all task parameters for workload filtering have been deprecated.

Table 18–1 Types of Advisor Task Parameters And Their Uses

Workload Filtering	Task Configuration	Schema Attributes	Recommendation Options
END_TIME	DAYS_TO_EXPIRE	DEF_INDEX_OWNER	ANALYSIS_SCOPE
INVALID_ACTION_LIST	JOURNALING	DEF_INDEX_TABLESPACE	COMPATIBILITY
INVALID_MODULE_LIST	REPORT_DATE_FORMAT	DEF_MVIEW_OWNER	CREATION_COST
INVALID_SQLSTRING_LIMIT		DEF_MVIEW_TABLESPACE	DML_VOLATILITY
INVALID_TABLE_LIST		DEF_MVLOG_TABLESPACE	LIMIT_PARTITION_ SCHEMES
INVALID_USERNAME_LIST		DEF_PARTITION_ TABLESPACE	MODE
RANKING_MEASURE		INDEX_NAME_TEMPLATE	PARTITIONING_TYPES
SQL_LIMIT		MVIEW_NAME_TEMPLATE	REFRESH_MODE
START_TIME			STORAGE_CHANGE
TIME_LIMIT			USE_SEPARATE_ TABLESPACES
VALID_ACTION_LIST			WORKLOAD_SCOPE
VALID_MODULE_LIST			
VALID_SQLSTRING_LIST			
VALID_TABLE_LIST			
VALID_USERNAME_LIST			

In the following example, set the storage change of task MYTASK to 100MB. This indicates 100MB of additional space for recommendations. A zero value would indicate that no additional space can be allocated. A negative value indicates that the advisor must attempt to trim the current space utilization by the specified amount.

```
EXECUTE DBMS_ADVISOR.SET_TASK_PARAMETER('MYTASK','STORAGE_CHANGE', 100000000);
```

In the following example, set the VALID_TABLE_LIST parameter to filter out all queries that do no consist of tables SH.SALES and SH.CUSTOMERS.

```
EXECUTE DBMS_ADVISOR.SET_TASK_PARAMETER ( -
   'MYTASK', 'VALID_TABLE_LIST', 'SH.SALES, SH.CUSTOMERS');
```

See *Oracle Database PL/SQL Packages and Types Reference* for more information regarding the SET_TASK_PARAMETER procedure and its parameters.

Evaluation Mode

The SQL Access Advisor operates in two modes: problem solving and evaluation. By default, SQL Access Advisor will attempt to solve access method problems by looking for enhancements to index structures, partitions, materialized views and materialized view logs. When doing evaluation only, SQL Access Advisor will only comment on what access structures the supplied workload will use. For example, a problem solving run may recommend creating a new index, adding a new column to a materialized view log, and so on, while an evaluation only scenario will only produce recommendations such as retaining an index, retaining a materialized view, and so on. The evaluation mode can be useful to see exactly which indexes and materialized views are actually being used by a workload.

View Intermediate Results During Recommendation Analysis

The SQL Access Advisor now has the ability to see intermediate results during the analysis operation. Previously, results of an analysis operation were unavailable until

the processing had completed or was interrupted by the user. Now, the user may access results in the corresponding recommendation and action tables even while the SQL Access Advisor task is still executing. The benefit is that long running tasks can provide evidence that may allow the user to accept the current results by interrupting the task rather than waiting for a lengthy execution to complete.

To accept the current set of recommendations, the user must interrupt the task. This will signal SQL Access Advisor to stop processing and mark the task as `INTERRUPTED`. At that point, the user may update recommendation attributes and generate scripts. Alternatively, the SQL Access Advisor can be allowed to complete the recommendation process.

Note that intermediate results represent recommendations for the workload contents up to that point in time. If it is critical that the recommendations be sensitive to the entire workload, then Oracle recommends that you allow the task execution to complete normally. Additionally, recommendations made by the advisor early in the recommendation process will not contain any base table partitioning recommendations because the partitioning analysis requires a substantial part of the workload to be processed before it can determine whether partitioning would be beneficial. Therefore, only later intermediate results will contain base table partitioning recommendations if the SQL Access Advisor detects a benefit.

Generating Recommendations

You can generate recommendations by using the `EXECUTE_TASK` procedure with your task name. After the procedure finishes, you can check the `DBA_ADVISOR_LOG` table for the actual execution status and the number of recommendations and actions that have been produced. The recommendations can be queried by task name in `{DBA, USER}_ADVISOR_RECOMMENDATIONS` and the actions for these recommendations can be viewed by task in `{DBA, USER}_ADVISOR_ACTIONS`.

EXECUTE_TASK Procedure This procedure performs the SQL Access Advisor analysis or evaluation for the specified task. Task execution is a synchronous operation, so control will not be returned to the user until the operation has completed, or a user-interrupt was detected. Upon return or execution of the task, you can check the `DBA_ADVISOR_LOG` table for the actual execution status.

Running `EXECUTE_TASK` generates recommendations, where a recommendation comprises one or more actions, such as creating a materialized view log and a materialized view. The syntax is as follows:

```
DBMS_ADVISOR.EXECUTE_TASK (task_name   IN VARCHAR2);
```

The following illustrates an example of using this procedure:

```
EXECUTE DBMS_ADVISOR.EXECUTE_TASK('MYTASK');
```

See *Oracle Database PL/SQL Packages and Types Reference* for more information regarding the `EXECUTE_TASK` procedure and its parameters.

Viewing Recommendations

Each recommendation generated by the SQL Access Advisor can be viewed using several catalog views, such as `(DBA, USER)_ADVISOR_RECOMMENDATIONS`. However, it is easier to use the `GET_TASK_SCRIPT` procedure or use the SQL Access Advisor in Enterprise Manager, which graphically displays the recommendations and provides hyperlinks to quickly see which SQL statements benefit from a recommendation. Each recommendation produced by the SQL Access Advisor is linked to the SQL statement it benefits.

The following shows the recommendation (rec_id) produced by an Advisor run, with their rank and total benefit. The rank is a measure of the importance of the queries that the recommendation helps. The benefit is the total improvement in execution cost (in terms of optimizer cost) of all the queries using the recommendation.

```
VARIABLE workload_name VARCHAR2(255);
VARIABLE task_name VARCHAR2(255);
EXECUTE :task_name := 'MYTASK';
EXECUTE :workload_name := 'MYWORKLOAD';

SELECT REC_ID, RANK, BENEFIT
FROM USER_ADVISOR_RECOMMENDATIONS WHERE TASK_NAME = :task_name;

    REC_ID       RANK    BENEFIT
---------- ---------- ----------
         1          2       2754
         2          3       1222
         3          1       5499
         4          4        594
```

To identify which query benefits from which recommendation, you can use the views DBA_* and USER_ADVISOR_SQLA_WK_STMTS. The precost and postcost numbers are in terms of the estimated optimizer cost (shown in EXPLAIN PLAN) without and with the recommended access structure changes, respectively. To see recommendations for each query, issue the following statement:

```
SELECT sql_id, rec_id, precost, postcost,
       (precost-postcost)*100/precost AS percent_benefit
FROM USER_ADVISOR_SQLA_WK_STMTS
WHERE TASK_NAME = :task_name AND workload_name = :workload_name;

    SQL_ID     REC_ID    PRECOST   POSTCOST PERCENT_BENEFIT
---------- ---------- ---------- ---------- ---------------
       121          1       3003        249      91.7082917
       122          2       1404        182       87.037037
       123          3       5503          4      99.9273124
       124          4        730        136       81.369863
```

Each recommendation consists of one or more actions, which must be implemented together to realize the benefit provided by the recommendation. The SQL Access Advisor produces the following types of actions:

- PARTITION BASE TABLE
- CREATE|DROP|RETAIN MATERIALIZED VIEW
- CREATE|ALTER|RETAIN MATERIALIZED VIEW LOG
- CREATE|DROP|RETAIN INDEX
- GATHER STATS

The PARTITION BASE TABLE action partitions an existing unpartitioned base table. The CREATE actions corresponds to new access structures. RETAIN recommendations indicate that existing access structures must be kept. DROP recommendations are only produced if the WORKLOAD_SCOPE parameter is set to FULL. The GATHER STATS action will generate a call to DBMS_STATS procedure to gather statistics on a newly generated access structure. Note that multiple recommendations may refer to the same action, however when generating a script for the recommendation, you will only see each action once.

In the following example, you can see how many distinct actions there are for this set of recommendations.

```
SELECT 'Action Count', COUNT(DISTINCT action_id) cnt
FROM USER_ADVISOR_ACTIONS WHERE task_name = :task_name;

'ACTIONCOUNT        CNT
------------   --------
Action Count         20

-- see the actions for each recommendations
SELECT rec_id, action_id, SUBSTR(command,1,30) AS command
FROM user_advisor_actions WHERE task_name = :task_name
ORDER BY rec_id, action_id;

    REC_ID  ACTION_ID COMMAND
---------- ---------- ------------------------------
         1          5 CREATE MATERIALIZED VIEW LOG
         1          6 ALTER MATERIALIZED VIEW LOG
         1          7 CREATE MATERIALIZED VIEW LOG
         1          8 ALTER MATERIALIZED VIEW LOG
         1          9 CREATE MATERIALIZED VIEW LOG
         1         10 ALTER MATERIALIZED VIEW LOG
         1         11 CREATE MATERIALIZED VIEW
         1         12 GATHER TABLE STATISTICS
         1         19 CREATE INDEX
         1         20 GATHER INDEX STATISTICS
         2          5 CREATE MATERIALIZED VIEW LOG
         2          6 ALTER MATERIALIZED VIEW LOG
         2          9 CREATE MATERIALIZED VIEW LOG
         ...
```

Each action has several attributes that pertain to the properties of the access structure. The name and tablespace for each access structure when applicable are placed in `attr1` and `attr2` respectively. The space occupied by each new access structure is in `num_attr1`. All other attributes are different for each action.

Table 18–2 maps SQL Access Advisor action information to the corresponding column in `DBA_ADVISOR_ACTIONS`. In the table, "MV" refers to a materialized view.

Table 18–2 SQL Access Advisor Action Attributes

	ATTR1	ATTR2	ATTR3	ATTR4	ATTR5	ATTR6	NUM_ ATTR1
CREATE INDEX	Index name	Index tablespace	Target table	BITMAP or BTREE	Index column list / expression	Unused	Storage size in bytes for the index
CREATE MATERIALIZED VIEW	MV name	MV tablespace	REFRESH COMPLETE REFRESH FAST, REFRESH FORCE, NEVER REFRESH	ENABLE QUERY REWRITE, DISABLE QUERY REWRITE	SQL SELECT statement	Unused	Storage size in bytes for the MV
CREATE MATERIALIZED VIEW LOG	Target table name	MV log tablespace	ROWID PRIMARY KEY, SEQUENCE OBJECT ID	INCLUDING NEW VALUES, EXCLUDING NEW VALUES	Table column list	Partitioning subclauses	Unused
CREATE REWRITE EQUIVALENCE	Name of equivalence	Checksum value	Unused	Unused	Source SQL statement	Equivalent SQL statement	Unused
DROP INDEX	Index name	Unused	Unused	Unused	Index columns	Unused	Storage size in bytes for the index
DROP MATERIALIZED VIEW	MV name	Unused	Unused	Unused	Unused	Unused	Storage size in bytes for the MV
DROP MATERIALIZED VIEW LOG	Target table name	Unused	Unused	Unused	Unused	Unused	Unused
PARTITION TABLE	Table name	RANGE, INTERVAL, LIST, HASH, RANGE-HASH, RANGE-LIST	Partition key for partitioning (column name or list of column names)	Partition key for subpartitioning (column name or list of column names)	SQL PARTITION clause	SQL SUBPARTITION clause	Unused
PARTITION INDEX	Index name	LOCAL, RANGE, HASH	Partition key for partitioning (list of column names)	Unused	SQL PARTITION clause	Unused	Unused
PARTITION ON MATERIALIZED VIEW	MV name	RANGE, INTERVAL, LIST, HASH, RANGE-HASH, RANGE-LIST	Partition key for partitioning (column name or list of column names)	Partition key for subpartitioning (column name or list of column names)	SQL SUBPARTITION clause	SQL SUBPARTITION clause	Unused

Table 18–2 (Cont.) SQL Access Advisor Action Attributes

	ATTR1	ATTR2	ATTR3	ATTR4	ATTR5	ATTR6	NUM_ ATTR1
RETAIN INDEX	Index name	Unused	Target table	BITMAP or BTREE	Index columns	Unused	Storage size in bytes for the index
RETAIN MATERIALIZED VIEW	MV name	Unused	REFRESH COMPLETE or REFRESH FAST	Unused	SQL SELECT statement	Unused	Storage size in bytes for the MV
RETAIN MATERIALIZED VIEW LOG	Target table name	Unused	Unused	Unused	Unused	Unused	Unused

The following PL/SQL procedure can be used to print out some of the attributes of the recommendations.

```
CONNECT SH/SH;
CREATE OR REPLACE PROCEDURE show_recm (in_task_name IN VARCHAR2) IS
CURSOR curs IS
  SELECT DISTINCT action_id, command, attr1, attr2, attr3, attr4
  FROM user_advisor_actions
  WHERE task_name = in_task_name
  ORDER BY action_id;
  v_action        number;
  v_command     VARCHAR2(32);
  v_attr1       VARCHAR2(4000);
  v_attr2       VARCHAR2(4000);
  v_attr3       VARCHAR2(4000);
  v_attr4       VARCHAR2(4000);
  v_attr5       VARCHAR2(4000);
BEGIN
  OPEN curs;
  DBMS_OUTPUT.PUT_LINE('========================================');
  DBMS_OUTPUT.PUT_LINE('Task_name = ' || in_task_name);
  LOOP
     FETCH curs INTO
       v_action, v_command, v_attr1, v_attr2, v_attr3, v_attr4 ;
   EXIT when curs%NOTFOUND;
   DBMS_OUTPUT.PUT_LINE('Action ID: ' || v_action);
   DBMS_OUTPUT.PUT_LINE('Command : ' || v_command);
   DBMS_OUTPUT.PUT_LINE('Attr1 (name)      : ' || SUBSTR(v_attr1,1,30));
   DBMS_OUTPUT.PUT_LINE('Attr2 (tablespace): ' || SUBSTR(v_attr2,1,30));
   DBMS_OUTPUT.PUT_LINE('Attr3             : ' || SUBSTR(v_attr3,1,30));
   DBMS_OUTPUT.PUT_LINE('Attr4             : ' || v_attr4);
   DBMS_OUTPUT.PUT_LINE('Attr5             : ' || v_attr5);
   DBMS_OUTPUT.PUT_LINE('----------------------------------------');
   END LOOP;
   CLOSE curs;
   DBMS_OUTPUT.PUT_LINE('=========END RECOMMENDATIONS============');
END show_recm;
/
-- see what the actions are using sample procedure
set serveroutput on size 99999
EXECUTE show_recm(:task_name);
A fragment of a sample output from this procedure is as follows:
Task_name = MYTASK
```

```
Action ID: 1
Command : CREATE MATERIALIZED VIEW LOG
Attr1 (name)      : "SH"."CUSTOMERS"
Attr2 (tablespace):
Attr3             : ROWID, SEQUENCE
Attr4             :  INCLUDING NEW VALUES
Attr5             :
----------------------------------------
..
----------------------------------------
Action ID: 15
Command : CREATE MATERIALIZED VIEW
Attr1 (name)      : "SH"."SH_MV$$_0004"
Attr2 (tablespace): "SH_MVIEWS"
Attr3             : REFRESH FAST WITH ROWID
Attr4             : ENABLE QUERY REWRITE
Attr5             :
----------------------------------------
..
----------------------------------------
Action ID: 19
Command : CREATE INDEX
Attr1 (name)      : "SH"."SH_IDX$$_0013"
Attr2 (tablespace): "SH_INDEXES"
Attr3             : "SH"."SH_MV$$_0002"
Attr4             : BITMAP
Attr5             :
```

See *Oracle Database PL/SQL Packages and Types Reference* for details regarding `Attr5` and `Attr6`.

Stopping the Recommendation Process

If the SQL Access Advisor takes too long to make its recommendations using the procedure `EXECUTE_TASK`, you can stop it by calling the `CANCEL_TASK` procedure and passing in the `task_name` for this recommendation process. If you use `CANCEL_TASK`, no recommendations will be made. Therefore, if recommendations are required, consider using the `INTERRUPT_TASK` procedure.

Interrupting Tasks The `INTERRUPT_TASK` procedure causes an Advisor operation to terminate as if it has reached its normal end. As a result, the user can see any recommendations that have been formed up to the point of the interrupt.

An interrupted task cannot be restarted. The syntax is as follows:

```
DBMS_ADVISOR.INTERRUPT_TASK (task_name IN VARCHAR2);
```

The following illustrates an example of using this procedure:

```
EXECUTE DBMS_ADVISOR.INTERRUPT_TASK ('MY_TASK');
```

Canceling Tasks The `CANCEL_TASK` procedure causes a currently executing operation to terminate. An Advisor operation may take a few seconds to respond to this request. Because all Advisor task procedures are synchronous, to cancel an operation, you must use a separate database session.

A cancel command effective restores the task to its condition prior to the start of the cancelled operation. Therefore, a cancelled task or data object cannot be restarted (but you can reset the task using `DBMS_ADVISOR.RESET_TASK` and then executing it again). Its syntax is as follows:

```
DBMS_ADVISOR.CANCEL_TASK (task_name   IN  VARCHAR2);
```

The following illustrates an example of using this procedure:

```
EXECUTE DBMS_ADVISOR.CANCEL_TASK('MYTASK');
```

See *Oracle Database PL/SQL Packages and Types Reference* for more information regarding the CANCEL_TASK procedure and its parameters.

Marking Recommendations

By default, all SQL Access Advisor recommendations are ready to be implemented, however, the user can choose to skip or exclude selected recommendations by using the MARK_RECOMMENDATION procedure. MARK_RECOMMENDATION allows the user to annotate a recommendation with a REJECT or IGNORE setting, which will cause the GET_TASK_SCRIPT to skip it when producing the implementation procedure. The syntax is as follows:

```
DBMS_ADVISOR.MARK_RECOMMENDATION (
   task_name          IN VARCHAR2
   id                 IN NUMBER,
   action             IN VARCHAR2);
```

The following example marks a recommendation with ID 2 as REJECT. This recommendation and any dependent recommendations will not appear in the script.

```
EXECUTE DBMS_ADVISOR.MARK_RECOMMENDATION('MYTASK', 2, 'REJECT');
```

If the Advisor made a partition recommendation (that is, a recommendation to partition one or multiple previously unpartitioned base tables), careful consideration should be given as to whether this recommendation should be skipped. The reason is that changing a table's partitioning scheme affects the cost of all queries, indexes, and materialized views defined on that table. Therefore, the Advisor's remaining recommendations on that table will no longer be optimal if you skip the partitioning recommendation. If you want to see recommendations on your workload that do not contain partitioning, you should reset the advisor task and rerun it with the ANALYSIS_SCOPE parameter changed to exclude partitioning recommendations.

See *Oracle Database PL/SQL Packages and Types Reference* for more information regarding the MARK_RECOMMENDATIONS procedure and its parameters.

Modifying Recommendations

Using the UPDATE_REC_ATTRIBUTES procedure, the SQL Access Advisor names and assigns ownership to new objects such as indexes and materialized views during the analysis operation. However, it does not necessarily choose appropriate names, so you may manually set the owner, name, and tablespace values for new objects. For recommendations referencing existing database objects, owner and name values cannot be changed. The syntax is as follows:

```
DBMS_ADVISOR.UPDATE_REC_ATTRIBUTES (
   task_name            IN VARCHAR2
   rec_id               IN NUMBER,
   action_id            IN NUMBER,
   attribute_name       IN VARCHAR2,
   value                IN VARCHAR2);
```

The attribute_name parameter can take the following values:

- OWNER

Specifies the owner name of the recommended object.

- NAME

  Specifies the name of the recommended object.

- TABLESPACE

  Specifies the tablespace of the recommended object.

The following example modifies the attribute TABLESPACE for recommendation ID 1, action ID 1 to SH_MVIEWS.

```
EXECUTE DBMS_ADVISOR.UPDATE_REC_ATTRIBUTES('MYTASK', 1, 1, -
                                           'TABLESPACE', 'SH_MVIEWS');
```

See *Oracle Database PL/SQL Packages and Types Reference* for more information regarding the UPDATE_REC_ATTRIBUTES procedure and its parameters.

Generating SQL Scripts

An alternative to querying the metadata to see the recommendations is to create a script of the SQL statements for the recommendations, using the procedure GET_TASK_SCRIPT. The resulting script is an executable SQL file that can contain DROP, CREATE, and ALTER statements. For new objects, the names of the materialized views, materialized view logs, and indexes are auto-generated by using the user-specified name template. You should review the generated SQL script before attempting to execute it.

There are several task parameters that control the naming conventions (MVIEW_NAME_TEMPLATE and INDEX_NAME_TEMPLATE), the owner for these new objects (DEF_INDEX_OWNER and DEF_MVIEW_OWNER), and the tablespaces (DEF_MVIEW_TABLESPACE and DEF_INDEX_TABLESPACE).

The following example shows how to generate a CLOB containing the script for the recommendations:

```
EXECUTE DBMS_ADVISOR.CREATE_FILE(DBMS_ADVISOR.GET_TASK_SCRIPT('MYTASK'),
               'ADVISOR_RESULTS', 'advscript.sql');
```

To save the script to a file, a directory path must be supplied so that the procedure CREATE_FILE knows where to store the script. In addition, read and write privileges must be granted on this directory. The following example shows how to save an advisor script CLOB to a file:

```
-- create a directory and grant permissions to read/write to it
CONNECT SH/SH;
CREATE DIRECTORY ADVISOR_RESULTS AS '/mydir';
GRANT READ ON DIRECTORY ADVISOR_RESULTS TO PUBLIC;
GRANT WRITE ON DIRECTORY ADVISOR_RESULTS TO PUBLIC;
```

The following is a fragment of a script generated by this procedure. The script also includes PL/SQL calls to gather statistics on the recommended access structures and marks the recommendations as IMPLEMENTED at the end:

```
Rem  Access Advisor V11.1.0.0.0 - Production
Rem
Rem  Username:        SH
Rem  Task:            MYTASK
Rem  Execution date:  15/08/2006 11:35
Rem
set feedback 1
set linesize 80
```

```
set trimspool on
set tab off
set pagesize 60
whenever sqlerror CONTINUE

CREATE MATERIALIZED VIEW LOG ON "SH"."PRODUCTS"
    WITH ROWID, SEQUENCE("PROD_ID","PROD_SUBCATEGORY")
    INCLUDING NEW VALUES;
ALTER MATERIALIZED VIEW LOG FORCE ON "SH"."PRODUCTS"
    ADD ROWID, SEQUENCE("PROD_ID","PROD_SUBCATEGORY")
    INCLUDING NEW VALUES;
..
CREATE MATERIALIZED VIEW "SH"."MV$$_00510002"
    REFRESH FAST WITH ROWID
    ENABLE QUERY REWRITE
    AS SELECT SH.CUSTOMERS.CUST_STATE_PROVINCE C1, COUNT(*) M1 FROM
SH.CUSTOMERS WHERE (SH.CUSTOMERS.CUST_STATE_PROVINCE = 'CA') GROUP
BY SH.CUSTOMERS.CUST_STATE_PROVINCE;
BEGIN
  DBMS_STATS.GATHER_TABLE_STATS('"SH"', '"MV$$_00510002"', NULL,
     DBMS_STATS.AUTO_SAMPLE_SIZE);
END;
/
..
CREATE BITMAP INDEX "SH"."MV$$_00510004_IDX$$_00510013"
    ON "SH"."MV$$_00510004" ("C4");
whenever sqlerror EXIT SQL.SQLCODE
BEGIN
  DBMS_ADVISOR.MARK_RECOMMENDATION('"MYTASK"',1,'IMPLEMENTED');
  DBMS_ADVISOR.MARK_RECOMMENDATION('"MYTASK"',2,'IMPLEMENTED');
  DBMS_ADVISOR.MARK_RECOMMENDATION('"MYTASK"',3,'IMPLEMENTED');
  DBMS_ADVISOR.MARK_RECOMMENDATION('"MYTASK"',4,'IMPLEMENTED');
END;
/
```

> **See Also:** *Oracle Database SQL Reference* for `CREATE DIRECTORY` syntax and *Oracle Database PL/SQL Packages and Types Reference* for detailed information about the `GET_TASK_SCRIPT` procedure

Special Considerations when Script Includes Partitioning Recommendations

The Advisor may recommend partitioning an existing unpartitioned base table to improve query performance. When the Advisor implementation script contains partition recommendations, you must take note of the following issues:

- Partitioning an existing table is a complex and extensive operation, which may take considerably longer than implementing a new index or materialized view. Sufficient time should be reserved for implementing this recommendation.
- While index and materialized view recommendations are easy to reverse by deleting the index or view, a table, once partitioned, cannot easily be restored to its original state. Therefore, you should ensure that you backup your database before executing a script containing partition recommendations.
- The Advisor invokes the `DBMS_REDEFINITION` package in order to implement partition recommendations. The package is called in such a way that it redefines an existing unpartitioned base table to be partitioned without requiring to shut down the database. However, if the table has bitmap indexes, they cannot be migrated properly. The user must then manually remove such indexes and replace them after the Advisor script has run successfully. If a table has such bitmap

indexes defined on it, the Advisor will contain an appropriate warning. In addition, you should carefully supervise the execution of the DBMS_REDEFINITION script to ensure it runs successfully.

- While repartitioning a base table, the DBMS_REDEFINITION package makes a temporary copy of the original table, which will occupy the same amount of space as the original table. Therefore, the repartitioning process requires sufficient free disk space for another copy of the largest table to be repartitioned. The user must ensure that such space is available before running the implementation script.
- If you decide not to implement a partition recommendation that the advisor has made, please note that all other recommendations on the same table in the same script (such as CREATE INDEX and CREATE MATERIALIZED VIEW recommendations) are dependent on the partitioning recommendation. In order to obtain accurate recommendations, you should not simply remove the partition recommendation from the script but rather rerun the advisor with partitioning disabled (for example, by setting parameter ANALYSIS_SCOPE to a value that does not include the keyword TABLE).

See Also: *Oracle Database SQL Reference* for CREATE DIRECTORY syntax and *Oracle Database PL/SQL Packages and Types Reference* for detailed information about the GET_TASK_SCRIPT procedure.

When Recommendations are no Longer Required

The RESET_TASK procedure resets a task to its initial starting point. This has the effect of removing all recommendations, and intermediate data from the task. The actual task status is set to INITIAL. The syntax is as follows:

```
DBMS_ADVISOR.RESET_TASK (task_name     IN VARCHAR2);
```

The following illustrates an example of using this procedure:

```
EXECUTE DBMS_ADVISOR.RESET_TASK('MYTASK');
```

See *Oracle Database PL/SQL Packages and Types Reference* for more information regarding the RESET_TASK procedure and its parameters.

Performing a Quick Tune

If you only want to tune a single SQL statement, the QUICK_TUNE procedure accepts as its input a task_name and a SQL statement. It will then create a task and workload and execute that task. There is no difference in the results from using QUICK_TUNE. They are exactly the same as those from using EXECUTE_TASK, but this approach is easier to use when there is only a single SQL statement to be tuned. The syntax is as follows:

```
DBMS_ADVISOR.QUICK_TUNE (
   advisor_name           IN VARCHAR2,
   task_name              IN VARCHAR2,
   attr1                  IN CLOB,
   attr2                  IN VARCHAR2 := NULL,
   attr3                  IN NUMBER := NULL,
   task_or_template       IN VARCHAR2 := NULL);
```

The following example shows how to quick tune a single SQL statement:

```
VARIABLE task_name VARCHAR2(255);
VARIABLE sql_stmt VARCHAR2(4000);
EXECUTE :sql_stmt := 'SELECT COUNT(*) FROM customers
```

```
                    WHERE cust_state_province =''CA''';
EXECUTE :task_name  := 'MY_QUICKTUNE_TASK';
EXECUTE DBMS_ADVISOR.QUICK_TUNE(DBMS_ADVISOR.SQLACCESS_ADVISOR,
             :task_name, :sql_stmt);
```

See *Oracle Database PL/SQL Packages and Types Reference* for more information regarding the QUICK_TUNE procedure and its parameters.

Managing Tasks

Every time recommendations are generated, tasks are created and, unless some maintenance is performed on these tasks, they will grow over time and will occupy storage space. There may be tasks that you want to keep and prevent accidental deletion. Therefore, there are several management operations that can be performed on tasks:

- Updating Task Attributes
- Deleting Tasks
- Setting the DAYS_TO_EXPIRE Parameter

Updating Task Attributes

Using the UPDATE_TASK_ATTRIBUTES procedure, you can:

- Change the name of a task.
- Give a task a description.
- Set the task to be read-only so it cannot be changed.
- Make the task a template upon which other tasks can be defined.
- Changes various attributes of a task or a task template.

The syntax is as follows:

```
DBMS_ADVISOR.UPDATE_TASK_ATTRIBUTES (
   task_name          IN VARCHAR2
   new_name           IN VARCHAR2 := NULL,
   description        IN VARCHAR2 := NULL,
   read_only          IN VARCHAR2 := NULL,
   is_template        IN VARCHAR2 := NULL,
   how_created        IN VARCHAR2 := NULL);
```

The following example updates the name of an task MYTASK to TUNING1:

```
EXECUTE DBMS_ADVISOR.UPDATE_TASK_ATTRIBUTES('MYTASK', 'TUNING1');
```

The following example marks the task TUNING1 to read-only

```
EXECUTE DBMS_ADVISOR.UPDATE_TASK_ATTRIBUTES('TUNING1', read_only => 'TRUE');
```

The following example marks the task MYTASK as a template.

```
EXECUTE DBMS_ADVISOR.UPDATE_TASK_ATTRIBUTES('TUNING1', is_template=>'TRUE');
```

See *Oracle Database PL/SQL Packages and Types Reference* for more information regarding the UPDATE_TASK_ATTRIBUTES procedure and its parameters.

Deleting Tasks

The DELETE_TASK procedure deletes existing Advisor tasks from the repository. The syntax is as follows:

```
DBMS_ADVISOR.DELETE_TASK (task_name  IN VARCHAR2);
```

The following illustrates an example of using this procedure:

```
EXECUTE DBMS_ADVISOR.DELETE_TASK('MYTASK');
```

See *Oracle Database PL/SQL Packages and Types Reference* for more information regarding the DELETE_TASK procedure and its parameters.

Setting the DAYS_TO_EXPIRE Parameter

When a task or workload object is created, the parameter DAYS_TO_EXPIRE is set to 30. The value indicates the number of days until the task or object will automatically be deleted by the system. If you wish to save a task or workload indefinitely, the DAYS_TO_EXPIRE parameter should be set to ADVISOR_UNLIMITED.

Using SQL Access Advisor Constants

You can use the constants shown in Table 18–3 with the SQL Access Advisor.

Table 18–3 SQL Access Advisor Constants

Constant	Description
ADVISOR_ALL	A value that is used to indicate all possible values. For string parameters, this value is equivalent to the wildcard % character.
ADVISOR_CURRENT	Indicates the current time or active set of elements. Typically, this is used in time parameters.
ADVISOR_DEFAULT	Indicates the default value. Typically used when setting task or workload parameters.
ADVISOR_UNLIMITED	A value that represents an unlimited numeric value.
ADVISOR_UNUSED	A value that represents an unused entity. When a parameter is set to ADVISOR_UNUSED, it will have no effect on the current operation. This is typically used for setting a parameter as unused for its dependent operations.
SQLACCESS_GENERAL	Specifies the name of a default SQL Access general-purpose task template. This template will set the DML_VOLATILITY task parameter to TRUE and ANALYSIS_SCOPE to INDEX, MVIEW.
SQLACCESS_OLTP	Specifies the name of a default SQL Access OLTP task template. This template will set the DML_VOLATILITY task parameter to TRUE and ANALYSIS_SCOPE to INDEX.
SQLACCESS_WAREHOUSE	Specifies the name of a default SQL Access warehouse task template. This template will set the DML_VOLATILITY task parameter to FALSE and EXECUTION_TYPE to INDEX, MVIEW.
SQLACCESS_ADVISOR	Contains the formal name of the SQL Access Advisor. It can be used when procedures require the Advisor name as an argument.

Examples of Using the SQL Access Advisor

This section illustrates some typical scenarios for using the SQL Access Advisor. Oracle Database provides a script that contains this chapter's examples, aadvdemo.sql.

Recommendations From a User-Defined Workload

The following example imports workload from a user-defined table, SH.USER_WORKLOAD. It then creates a task called MYTASK, sets the storage budget to 100 MB and runs the task. The recommendations are printed out using a PL/SQL procedure. Finally, it generates a script, which can be used to implement the recommendations.

Step 1 Prepare the USER_WORKLOAD table

The USER_WORKLOAD table is loaded with SQL statements as follows:

```
CONNECT SH/SH;
-- aggregation with selection
INSERT INTO user_workload (username, module, action, priority, sql_text)
VALUES ('SH', 'Example1', 'Action', 2,
'SELECT   t.week_ending_day, p.prod_subcategory,
          SUM(s.amount_sold) AS dollars, s.channel_id, s.promo_id
 FROM sales s, times t, products p WHERE s.time_id = t.time_id
 AND s.prod_id = p.prod_id AND s.prod_id > 10 AND s.prod_id < 50
 GROUP BY t.week_ending_day, p.prod_subcategory,
          s.channel_id, s.promo_id')
/

-- aggregation with selection
INSERT INTO user_workload (username, module, action, priority, sql_text)
VALUES ('SH', 'Example1', 'Action', 2,
 'SELECT   t.calendar_month_desc, SUM(s.amount_sold) AS dollars
  FROM     sales s , times t
  WHERE    s.time_id = t.time_id
  AND    s.time_id between TO_DATE(''01-JAN-2000'', ''DD-MON-YYYY'')
                       AND TO_DATE(''01-JUL-2000'', ''DD-MON-YYYY'')
GROUP BY t.calendar_month_desc')
/

--Load all SQL queries.
INSERT INTO user_workload (username, module, action, priority, sql_text)
VALUES ('SH', 'Example1', 'Action', 2,
'SELECT ch.channel_class, c.cust_city, t.calendar_quarter_desc,
   SUM(s.amount_sold) sales_amount
FROM sales s, times t, customers c, channels ch
WHERE s.time_id = t.time_id AND s.cust_id = c.cust_id
AND s.channel_id = ch.channel_id AND c.cust_state_province = ''CA''
AND   ch.channel_desc IN (''Internet'',''Catalog'')
AND   t.calendar_quarter_desc IN (''1999-Q1'',''1999-Q2'')
GROUP BY ch.channel_class, c.cust_city, t.calendar_quarter_desc')
/

-- order by
INSERT INTO user_workload (username, module, action, priority, sql_text)
VALUES ('SH', 'Example1', 'Action', 2,
  'SELECT c.country_id, c.cust_city, c.cust_last_name
FROM customers c WHERE c.country_id IN (52790, 52789)
ORDER BY c.country_id, c.cust_city, c.cust_last_name')
/
COMMIT;

CONNECT SH/SH;
set serveroutput on;

VARIABLE task_id NUMBER;
VARIABLE task_name VARCHAR2(255);
VARIABLE workload_name VARCHAR2(255);
```

Step 2 Create a SQL Tuning Set named MYWORKLOAD

```
EXECUTE :workload_name := 'MYWORKLOAD';
EXECUTE DBMS_SQLTUNE.CREATE_SQLSET(:workload_name, 'test purposeV);
```

Step 3 Load the SQL Tuning Set from the user-defined table SH.USER_WORKLOAD

```
DECLARE
  sqlset_cur DBMS_SQLTUNE.SQLSET_CURSOR;    /*a sqlset cursor variable*/
BEGIN
OPEN  sqlset_cur FOR
  SELECT
    SQLSET_ROW(null, sql_text, null, null, username, null,
     null, 0,0,0,0,0,0,0,0,null, 0,0,0,0)
   AS ROW
  FROM USER_WORKLOAD;
DBMS_SQLTUNE.LOAD_SQLSET(:workload_name, sqlset_cur);
END;
```

Step 4 Create a task named MYTASK

```
EXECUTE :task_name := 'MYTASK';
EXECUTE DBMS_ADVISOR.CREATE_TASK('SQL Access Advisor', :task_id, :task_name);
```

Step 5 Set task parameters

```
EXECUTE DBMS_ADVISOR.SET_TASK_PARAMETER(:task_name, 'STORAGE_CHANGE', 100);
EXECUTE DBMS_ADVISOR.SET_TASK_PARAMETER(:task_name, 'ANALYSIS_SCOPE, INDEX');
```

Step 6 Create a link between the SQL Tuning Set and the task

```
EXECUTE DBMS_ADVISOR.ADD_STS_REF(:task_name, :workload_name);
```

Step 7 Execute the task

```
EXECUTE DBMS_ADVISOR.EXECUTE_TASK(:task_name);
```

Step 8 View the recommendations

```
-- See the number of recommendations and the status of the task.
SELECT rec_id, rank, benefit
FROM user_advisor_recommendations WHERE task_name = :task_name;
```

See "Viewing Recommendations" on page 18-12 or "Generating SQL Scripts" on page 18-19 for further details.

```
-- See recommendation for each query.
SELECT sql_id, rec_id, precost, postcost,
      (precost-postcost)*100/precost AS percent_benefit
FROM user_advisor_sqla_wk_stmts
WHERE task_name = :task_name AND workload_name = :workload_name;

-- See the actions for each recommendations.
SELECT rec_id, action_id, SUBSTR(command,1,30) AS command
FROM user_advisor_actions
WHERE task_name = :task_name
ORDER BY rec_id, action_id;

-- See what the actions are using sample procedure.
SET SERVEROUTPUT ON SIZE 99999
EXECUTE show_recm(:task_name);
```

Step 9 Generate a script to Implement the recommendations

```
EXECUTE DBMS_ADVISOR.CREATE_FILE(DBMS_ADVISOR.GET_TASK_SCRIPT(:task_name),
                                 'ADVISOR_RESULTS', 'Example1_script.sql');
```

Generate Recommendations Using a Task Template

The following example creates a template and then uses it to create a task. It then uses this task to generate recommendations from a user-defined table, similar to "Recommendations From a User-Defined Workload" on page 18-23.

```
CONNECT SH/SH;
VARIABLE template_id NUMBER;
VARIABLE template_name VARCHAR2(255);
```

Step 1 Create a template called MY_TEMPLATE

```
EXECUTE :template_name := 'MY_TEMPLATE';
EXECUTE DBMS_ADVISOR.CREATE_TASK ( -
   'SQL Access Advisor',:template_id, :template_name, is_template=>'TRUE');
```

Step 2 Set template parameters

Set naming conventions for recommended indexes and materialized views.

```
EXECUTE DBMS_ADVISOR.SET_TASK_PARAMETER ( -
   :template_name,  'INDEX_NAME_TEMPLATE', 'SH_IDX$$_<SEQ>');
EXECUTE DBMS_ADVISOR.SET_TASK_PARAMETER ( -
   :template_name, 'MVIEW_NAME_TEMPLATE', 'SH_MV$$_<SEQ>');

--Set default owners for recommended indexes/materialized views.
EXECUTE DBMS_ADVISOR.SET_TASK_PARAMETER ( -
   :template_name, 'DEF_INDEX_OWNER', 'SH');
EXECUTE DBMS_ADVISOR.SET_TASK_PARAMETER ( -
   :template_name, 'DEF_MVIEW_OWNER', 'SH');

--Set default tablespace for recommended indexes/materialized views.
EXECUTE DBMS_ADVISOR.SET_TASK_PARAMETER ( -
   :template_name, 'DEF_INDEX_TABLESPACE', 'SH_INDEXES');
EXECUTE DBMS_ADVISOR.SET_TASK_PARAMETER ( -
   :template_name, 'DEF_MVIEW_TABLESPACE', 'SH_MVIEWS');
```

Step 3 Create a task using the template

```
VARIABLE task_id NUMBER;
VARIABLE task_name VARCHAR2(255);
EXECUTE :task_name := 'MYTASK';
EXECUTE DBMS_ADVISOR.CREATE_TASK ( -
   'SQL Access Advisor', :task_id, :task_name, template => 'MY_TEMPLATE');

--See the parameter settings for task
SELECT parameter_name, parameter_value
FROM user_advisor_parameters
WHERE task_name = :task_name AND (parameter_name LIKE '%MVIEW%'
   OR parameter_name LIKE '%INDEX%');
```

Step 4 Create a SQL Tuning Set named MYWORKLOAD

```
EXECUTE :workload_name := 'MYWORKLOAD';
EXECUTE DBMS_SQLTUNE.CREATE_SQLSET(:workload_name, 'test_purpose');
```

Step 5 Load the SQL Tuning Set from the user-defined table SH.USER_WORKLOAD

```
DECLARE
   sqlset_cur DBMS_SQLTUNE.SQLSET_CURSOR;  /*a sqlset cursor variable*/
BEGIN
OPEN sqlset_cur FOR
   SELECT
```

```
   SQLSET_ROW(null,sql_text,null,null,username, null, null, 0,0,0,0,0,0,0,0,0,
      null,0,0,00) AS row
   FROM user_workload;
DBMS_SQLTUNE.LOAD_SQLSET(:workload_name, sqlsetcur);
END;
```

Step 6 Create a link between the workload and the task

```
EXECUTE DBMS_ADVISOR.ADD_STS_REF(:task_name, :workload_name);
```

Step 7 Execute the task

```
EXECUTE DBMS_ADVISOR.EXECUTE_TASK(:task_name);
```

Step 8 Generate a script

```
EXECUTE DBMS_ADVISOR.CREATE_FILE(DBMS_ADVISOR.GET_TASK_SCRIPT(:task_name),-
                                 'ADVISOR_RESULTS', 'Example2_script.sql');
```

Evaluate Current Usage of Indexes and Materialized Views

This example illustrates how the SQL Access Advisor can be used to evaluate the utilization of existing indexes and materialized views. We assume the workload is loaded into `USER_WORKLOAD` table as in "Recommendations From a User-Defined Workload" on page 18-23. The indexes and materialized views that are being currently used (by the given workload) will appear as `RETAIN` actions in the SQL Access Advisor recommendations.

```
VARIABLE task_id NUMBER;
VARIABLE task_name VARCHAR2(255);
VARIABLE workload_name VARCHAR2(255);
```

Step 1 Create a SQL Tuning Set named WORKLOAD

```
EXECUTE :workload_name := 'MYWORKLOAD';
EXECUTE DBMS_SQLTUNE.CREATE_SQLSET(:workload_name, 'test_purpose');
```

Step 2 Load the SQL Tuning Set from the user-defined table SH.USER_WORKLOAD

```
DECLARE
  sqlset_cur DBMS_SQLTUNE.SQLSET_CURSOR;  /*a sqlset cursor variable*/
BEGIN
OPEN sqlset_cur FOR
SELECT
  SQLSET_ROW(null,sql_text,null,null,username, null, null, 0,0,0,0,0,0,0,0,0,
   null, 0,0,0,0)
    AS ROW
  FROM user_workload;
DBMS_SQLTUNE.LOAD_SQLSET(:workload_name, :sqlsetcur);
END;
```

Step 3 Create a task named MY_EVAL_TASK

```
EXECUTE :task_name := 'MY_EVAL_TASK';
EXECUTE DBMS_ADVISOR.CREATE_TASK ('SQL Access Advisor', :task_id, :task_name);
```

Step 4 Create a link between workload and task

```
EXECUTE DBMS_ADVISOR.ADD_STS_REF(:task_name, :workload_name);
```

Step 5 Set task parameters to indicate EVALUATION ONLY task

```
EXECUTE DBMS_ADVISOR.SET_TASK_PARAMETER (:task_name, 'EVALUATION_ONLY', 'TRUE');
```

Step 6 Execute the task

```
EXECUTE DBMS_ADVISOR.EXECUTE_TASK(:task_name);
```

Step 7 View evaluation results

```
--See the number of recommendations and the status of the task.
SELECT rec_id, rank, benefit
FROM user_advisor_recommendations WHERE task_name = :task_name;

--See the actions for each recommendation.
SELECT rec_id, action_id, SUBSTR(command,1,30) AS command, attr1 AS name
FROM user_advisor_actions WHERE task_name = :task_name
ORDER BY rec_id, action_id;
```

Tuning Materialized Views for Fast Refresh and Query Rewrite

Several DBMS_MVIEW procedures can help you create materialized views that are optimized for fast refresh and query rewrite. The EXPLAIN_MVIEW procedure can tell you whether a materialized view is fast refreshable or eligible for general query rewrite and EXPLAIN_REWRITE will tell you whether query rewrite will occur. However, neither tells you how to achieve fast refresh or query rewrite.

To further facilitate the use of materialized views, the TUNE_MVIEW procedure shows you how to optimize your CREATE MATERIALIZED VIEW statement and to meet other requirements such as materialized view log and rewrite equivalence relationship for fast refresh and general query rewrite. TUNE_MVIEW analyzes and processes the CREATE MATERIALIZED VIEW statement and generates two sets of output results: one for the materialized view implementation and the other for undoing the CREATE MATERIALIZED VIEW operations. The two sets of output results can be accessed through views or be stored in external script files created by the SQL Access Advisor. These external script files are ready to execute to implement the materialized view.

With the TUNE_MVIEW procedure, you no longer require a detailed understanding of materialized views to create a materialized view in an application because the materialized view and its required components (such as a materialized view log) will be created correctly through the procedure.

See *Oracle Database PL/SQL Packages and Types Reference* for detailed information about the TUNE_MVIEW procedure.

DBMS_ADVISOR.TUNE_MVIEW Procedure

This section discusses the following information:

- TUNE_MVIEW Syntax and Operations
- Accessing TUNE_MVIEW Output Results

TUNE_MVIEW Syntax and Operations

The syntax for TUNE_MVIEW is as follows:

```
DBMS_ADVISOR.TUNE_MVIEW (
  task_name IN OUT VARCHAR2, mv_create_stmt IN [CLOB | VARCHAR2])
```

The TUNE_MVIEW procedure takes two input parameters: task_name and mv_create_stmt. task_name is a user-provided task identifier used to access the output results. mv_create_stmt is a complete CREATE MATERIALIZED VIEW statement that is to be tuned. If the input CREATE MATERIALIZED VIEW statement does not have the clauses of REFRESH FAST or ENABLE QUERY REWRITE, or both,

TUNE_MVIEW will use the default clauses REFRESH FORCE and DISABLE QUERY REWRITE to tune the statement to be fast refreshable if possible or only complete refreshable otherwise.

The TUNE_MVIEW procedure handles a broad range of CREATE MATERIALIZED VIEW statements that can have arbitrary defining queries in them. The defining query could be a simple SELECT statement or a complex query with set operators or inline views. When the defining query of the materialized view contains the clause REFRESH FAST, TUNE_MVIEW analyzes the query and checks to see if it is fast refreshable. If it is already fast refreshable, the procedure will return a message saying "the materialized view is already optimal and cannot be further tuned". Otherwise, the TUNE_MVIEW procedure will start the tuning work on the given statement.

The TUNE_MVIEW procedure can generate the output statements that correct the defining query by adding extra columns such as required aggregate columns or fix the materialized view logs so that FAST REFRESH is possible. In the case of a complex defining query, the TUNE_MVIEW procedure may decompose the query and generates two or more fast refreshable materialized views or will restate the materialized view in a way to fulfill fast refresh requirements as much as possible. The TUNE_MVIEW procedure supports defining queries with the following complex query constructs:

- Set operators (UNION, UNION ALL, MINUS, and INTERSECT)
- COUNT DISTINCT
- SELECT DISTINCT
- Inline views

When the ENABLE QUERY REWRITE clause is specified, TUNE_MVIEW will also fix the statement using a process similar to REFRESH FAST, that will redefine the materialized view so that as many of the advanced forms of query rewrite are possible.

The TUNE_MVIEW procedure generates two sets of output results as executable statements. One set of the output (IMPLEMENTATION) is for implementing materialized views and required components such as materialized view logs or rewrite equivalences to achieve fast refreshability and query rewritablity as much as possible. The other set of the output (UNDO) is for dropping the materialized views and the rewrite equivalences in case you decide they are not required.

The output statements for the IMPLEMENTATION process include:

- CREATE MATERIALIZED VIEW LOG statements: creates any missing materialized view logs required for fast refresh.
- ALTER MATERIALIZED VIEW LOG FORCE statements: fixes any materialized view log related requirements such as missing filter columns, sequence, and so on, required for fast refresh.
- One or more CREATE MATERIALIZED VIEW statements: In the case of one output statement, the original defining query is directly restated and transformed. Simple query transformation could be just adding required columns. For example, add rowid column for materialized join view and add aggregate column for materialized aggregate view. In the case of decomposition, multiple CREATE MATERIALIZED VIEW statements are generated and form a nested materialized view hierarchy in which one or more submaterialized views are referenced by a new top-level materialized view modified from the original statement. This is to achieve fast refresh and query rewrite as much as possible. Submaterialized views are often fast refreshable.

- BUILD_SAFE_REWRITE_EQUIVALENCE statement: enables the rewrite of top-level materialized views using submaterialized views. It is required to enable query rewrite when a composition occurs.

Note that the decomposition result implies no sharing of submaterialized views. That is, in the case of decomposition, the TUNE_MVIEW output will always contain new submaterialized view and it will not reference existing materialized views.

The output statements for the UNDO process include:

- DROP MATERIALIZED VIEW statements to reverse the materialized view creations (including submaterialized views) in the IMPLEMENTATION process.
- DROP_REWRITE_EQUIVALENCE statement to remove the rewrite equivalence relationship built in the IMPLEMENTATION process if needed.

Note that the UNDO process does not include statement to drop materialized view logs. This is because materialized view logs can be shared by many different materialized views, some of which may reside on remote Oracle instances.

Accessing TUNE_MVIEW Output Results

There are two ways to access TUNE_MVIEW output results:

- Script generation using DBMS_ADVISOR.GET_TASK_SCRIPT function and DBMS_ADVISOR.CREATE_FILE procedure.
- Use USER_TUNE_MVIEW or DBA_TUNE_MVIEW views.

USER_TUNE_MVIEW and DBA_TUNE_MVIEW Views After executing TUNE_MVIEW, the results are output into the SQL Access Advisor repository tables and are accessible through the Oracle views, USER_TUNE_MVIEW and DBA_TUNE_MVIEW. See *Oracle Database Reference* for further details.

Script Generation DBMS_ADVISOR Function and Procedure The most straightforward method for generating the execution scripts for a recommendation is to use the procedure DBMS_ADVISOR.GET_TASK_SCRIPT. The following is a simple example. First, a directory must be defined which is where the results will be stored:

```
CREATE DIRECTORY TUNE_RESULTS AS  '/tmp/script_dir';
GRANT READ, WRITE ON DIRECTORY TUNE_RESULTS TO PUBLIC;
```

Now generate both the implementation and undo scripts and place them in /tmp/script_dir/mv_create.sql and /tmp/script_dir/mv_undo.sql, respectively.

```
EXECUTE DBMS_ADVISOR.CREATE_FILE(DBMS_ADVISOR.GET_TASK_SCRIPT(:task_name),-
      'TUNE_RESULTS', 'mv_create.sql');
EXECUTE DBMS_ADVISOR.CREATE_FILE(DBMS_ADVISOR.GET_TASK_SCRIPT(:task_name, -
      'UNDO'), 'TUNE_RESULTS', 'mv_undo.sql');
```

Now let us review some examples using the TUNE_MVIEW procedure.

Example 18–1 Optimizing the Defining Query for Fast Refresh

This example shows how TUNE_MVIEW changes the defining query to be fast refreshable. A CREATE MATERIALIZED VIEW statement is defined in variable create_mv_ddl, which includes a FAST REFRESH clause. Its defining query contains a single query block in which an aggregate column, SUM(s.amount_sold), does not have the required aggregate columns to support fast refresh. If you execute the TUNE_

MVIEW statement with this MATERIALIZED VIEW CREATE statement, the resulting materialized view recommendation will be fast refreshable:

```
VARIABLE task_cust_mv VARCHAR2(30);
VARIABLE create_mv_ddl VARCHAR2(4000);
EXECUTE :task_cust_mv := 'cust_mv';

EXECUTE :create_mv_ddl := '
CREATE MATERIALIZED VIEW cust_mv
REFRESH FAST
DISABLE QUERY REWRITE AS
SELECT s.prod_id, s.cust_id, SUM(s.amount_sold) sum_amount
FROM sales s, customers cs
WHERE s.cust_id = cs.cust_id
GROUP BY s.prod_id, s.cust_id';

EXECUTE DBMS_ADVISOR.TUNE_MVIEW(:task_cust_mv, :create_mv_ddl);
```

The original defining query of cust_mv has been modified by adding aggregate columns in order to be fast refreshable.

The output from TUNE_MVIEW includes an optimized materialized view defining query as follows:

```
CREATE MATERIALIZED VIEW SH.CUST_MV
REFRESH FAST WITH ROWID
DISABLE QUERY REWRITE AS
SELECT SH.SALES.PROD_ID C1, SH.CUSTOMERS.CUST_ID C2,
          SUM("SH"."SALES"."AMOUNT_SOLD") M1,
          COUNT("SH"."SALES"."AMOUNT_SOLD") M2,
          COUNT(*) M3
     FROM SH.SALES, SH.CUSTOMERS
     WHERE SH.CUSTOMERS.CUST_ID = SH.SALES.CUST_ID
     GROUP BY SH.SALES.PROD_ID, SH.CUSTOMERS.CUST_ID;
```

The UNDO output is as follows:

```
DROP MATERIALIZED VIEW SH.CUST_MV;
```

Example 18–2 Access IMPLEMENTATION Output Through USER_TUNE_MVIEW View

```
SELECT STATEMENT FROM USER_TUNE_MVIEW
WHERE TASK_NAME= :task_cust_mv AND SCRIPT_TYPE='IMPLEMENTATION';
```

Example 18–3 Save IMPLEMENTATION Output in a Script File

```
CREATE DIRECTORY TUNE_RESULTS AS '/myscript'
GRANT READ, WRITE ON DIRECTORY TUNE_RESULTS TO PUBLIC;

EXECUTE DBMS_ADVISOR.CREATE_FILE(DBMS_ADVISOR.GET_TASK_SCRIPT(:task_cust_mv), -
   'TUNE_RESULTS', 'mv_create.sql');
```

Example 18–4 Enable Query Rewrite by Creating Multiple Materialized Views

This example shows how a materialized view's defining query with set operators UNION, which is not supported by query rewrite, can be decomposed into a number of submaterialized views and then query rewrite is possible. The input detail tables are assumed to be sales, customers, and countries, and they do not have materialized view logs.

First, you need to execute the TUNE_MVIEW statement with the CREATE MATERIALIZED VIEW statement defined in the variable create_mv_ddl.

```
EXECUTE :task_cust_mv := 'cust_mv2';

EXECUTE :create_mv_ddl := '
CREATE MATERIALIZED VIEW cust_mv
ENABLE QUERY REWRITE AS
SELECT s.prod_id, s.cust_id, COUNT(*) cnt, SUM(s.amount_sold) sum_amount
FROM sales s, customers cs, countries cn
WHERE s.cust_id = cs.cust_id AND cs.country_id = cn.country_id
AND cn.country_name IN (''USA'',''Canada'')
GROUP BY s.prod_id, s.cust_id
UNION
SELECT s.prod_id, s.cust_id, COUNT(*) cnt, SUM(s.amount_sold) sum_amount
FROM sales s, customers cs
WHERE s.cust_id = cs.cust_id AND s.cust_id IN (1005,1010,1012)
GROUP BY s.prod_id, s.cust_id';
```

The materialized view defining query contains a `UNION` set operator that does not support general query rewrite but if it is decomposed into multiple materialized views, query rewrite is possible. In order to support general query rewrite, the `MATERIALIZED VIEW` defining query will be decomposed.

```
EXECUTE DBMS_ADVISOR.TUNE_MVIEW(:task_cust_mv, :create_mv_ddl);
```

The following recommendation from `TUNE_MVIEW` is comprised of the materialized view logs and multiple materialized view:

```
CREATE MATERIALIZED VIEW LOG ON "SH"."CUSTOMERS"
WITH ROWID, SEQUENCE("CUST_ID")
    INCLUDING NEW VALUES;

ALTER MATERIALIZED VIEW LOG FORCE ON
    "SH"."CUSTOMERS"
    ADD ROWID, SEQUENCE("CUST_ID")
    INCLUDING NEW VALUES;

CREATE MATERIALIZED VIEW LOG ON
    "SH"."SALES"
    WITH ROWID, SEQUENCE("PROD_ID","CUST_ID","AMOUNT_SOLD")
    INCLUDING NEW VALUES;

ALTER MATERIALIZED VIEW LOG FORCE ON
    "SH"."SALES"
    ADD ROWID, SEQUENCE("PROD_ID","CUST_ID","AMOUNT_SOLD")
    INCLUDING NEW VALUES;

CREATE MATERIALIZED VIEW LOG ON
    "SH"."COUNTRIES"
    WITH ROWID, SEQUENCE("COUNTRY_ID","COUNTRY_NAME")
    INCLUDING NEW VALUES;

ALTER MATERIALIZED VIEW LOG FORCE ON
    "SH"."COUNTRIES"
    ADD ROWID, SEQUENCE("COUNTRY_ID","COUNTRY_NAME")
    INCLUDING NEW VALUES;

ALTER MATERIALIZED VIEW LOG FORCE ON
    "SH"."CUSTOMERS"
    ADD ROWID, SEQUENCE("CUST_ID","COUNTRY_ID")
    INCLUDING NEW VALUES;

ALTER MATERIALIZED VIEW LOG FORCE ON
```

```
    "SH"."SALES"
    ADD ROWID, SEQUENCE("PROD_ID","CUST_ID","AMOUNT_SOLD")
    INCLUDING NEW VALUES;

CREATE MATERIALIZED VIEW SH.CUST_MV$SUB1
    REFRESH FAST WITH ROWID ON COMMIT
    ENABLE QUERY REWRITE
    AS SELECT SH.SALES.PROD_ID C1, SH.CUSTOMERS.CUST_ID C2,
 SUM("SH"."SALES"."AMOUNT_SOLD")
        M1, COUNT("SH"."SALES"."AMOUNT_SOLD") M2, COUNT(*) M3 FROM SH.SALES,
        SH.CUSTOMERS WHERE SH.CUSTOMERS.CUST_ID = SH.SALES.CUST_ID AND
 (SH.SALES.CUST_ID IN (1012, 1010, 1005))
GROUP BY SH.SALES.PROD_ID, SH.CUSTOMERS.CUST_ID;

CREATE MATERIALIZED VIEW SH.CUST_MV$SUB2
    REFRESH FAST WITH ROWID ON COMMIT
    ENABLE QUERY REWRITE
    AS SELECT SH.SALES.PROD_ID C1, SH.CUSTOMERS.CUST_ID C2,
      SH.COUNTRIES.COUNTRY_NAME  C3, SUM("SH"."SALES"."AMOUNT_SOLD") M1,
COUNT("SH"."SALES".
"AMOUNT_SOLD")
        M2, COUNT(*) M3 FROM SH.SALES, SH.CUSTOMERS, SH.COUNTRIES WHERE
 SH.CUSTOMERS.CUST_ID
        = SH.SALES.CUST_ID AND SH.COUNTRIES.COUNTRY_ID = SH.CUSTOMERS.COUNTRY_ID
        AND (SH.COUNTRIES.COUNTRY_NAME IN ('USA', 'Canada')) GROUP BY
 SH.SALES.PROD_ID,
        SH.CUSTOMERS.CUST_ID, SH.COUNTRIES.COUNTRY_NAME;

CREATE MATERIALIZED VIEW SH.CUST_MV
    REFRESH FORCE WITH ROWID
    ENABLE QUERY REWRITE
    AS  (SELECT "CUST_MV$SUB2"."C1" "PROD_ID","CUST_MV$SUB2"."C2"
"CUST_ID",SUM("CUST_MV$SUB2"."M3")
        "CNT",SUM("CUST_MV$SUB2"."M1") "SUM_AMOUNT" FROM "SH"."CUST_MV$SUB2"
        "CUST_MV$SUB2" GROUP BY "CUST_MV$SUB2"."C1","CUST_MV$SUB2"."C2")UNION
        (SELECT "CUST_MV$SUB1"."C1" "PROD_ID","CUST_MV$SUB1"."C2"
"CUST_ID",SUM("CUST_MV$SUB1"."M3")
        "CNT",SUM("CUST_MV$SUB1"."M1") "SUM_AMOUNT" FROM "SH"."CUST_MV$SUB1"
        "CUST_MV$SUB1" GROUP BY "CUST_MV$SUB1"."C1","CUST_MV$SUB1"."C2");

BEGIN
DBMS_ADVANCED_REWRITE.BUILD_SAFE_REWRITE_EQUIVALENCE ('SH.CUST_MV$RWEQ',
  'SELECT s.prod_id, s.cust_id, COUNT(*) cnt,
          SUM(s.amount_sold) sum_amount
   FROM sales s, customers cs, countries cn
   WHERE s.cust_id = cs.cust_id AND cs.country_id = cn.country_id
         AND cn.country_name IN (''USA'',''Canada'')
   GROUP BY s.prod_id, s.cust_id
   UNION
   SELECT s.prod_id, s.cust_id, COUNT(*) cnt,
          SUM(s.amount_sold) sum_amount
   FROM sales s, customers cs
   WHERE s.cust_id = cs.cust_id AND s.cust_id IN (1005,1010,1012)
   GROUP BY s.prod_id, s.cust_id',
  '(SELECT "CUST_MV$SUB2"."C3" "PROD_ID","CUST_MV$SUB2"."C2" "CUST_ID",
          SUM("CUST_MV$SUB2"."M3") "CNT",
          SUM("CUST_MV$SUB2"."M1") "SUM_AMOUNT"
    FROM "SH"."CUST_MV$SUB2" "CUST_MV$SUB2"
    GROUP BY "CUST_MV$SUB2"."C3","CUST_MV$SUB2"."C2")
   UNION
```

```
   (SELECT "CUST_MV$SUB1"."C2" "PROD_ID","CUST_MV$SUB1"."C1" "CUST_ID",
           "CUST_MV$SUB1"."M3" "CNT","CUST_MV$SUB1"."M1" "SUM_AMOUNT"
    FROM "SH"."CUST_MV$SUB1" "CUST_MV$SUB1")',-1553577441)
END;
/;
```

The DROP output is as follows:

```
DROP MATERIALIZED VIEW SH.CUST_MV$SUB1
DROP MATERIALIZED VIEW SH.CUST_MV$SUB2
DROP MATERIALIZED VIEW SH.CUST_MV
DBMS_ADVANCED_REWRITE.DROP_REWRITE_EQUIVALENCE('SH.CUST_MV$RWEQ')
```

The original defining query of cust_mv has been decomposed into two submaterialized views seen as cust_mv$SUB1 and cust_mv$SUB2. One additional column COUNT(amount_sold) has been added in cust_mv$SUB1 to make that materialized view fast refreshable.

The original defining query of cust_mv has been modified to query the two submaterialized views instead where both submaterialized views are fast refreshable and support general query rewrite.

The required materialized view logs are added to enable fast refresh of the submaterialized views. Note that, for each detail table, two materialized view log statements are generated: one is the CREATE MATERIALIZED VIEW statement and the other is an ALTER MATERIALIZED VIEW FORCE statement. This is to ensure the CREATE script can be run multiple times.

The BUILD_SAFE_REWRITE_EQUIVALENCE statement is to connect the old defining query to the defining query of the new top-level materialized view. It is to ensure that query rewrite will make use of the new top-level materialized view to answer the query.

Example 18–5 Access IMPLEMENTATION Output Through USER_TUNE_MVIEW View

```
SELECT * FROM USER_TUNE_MVIEW
WHERE TASK_NAME='cust_mv2'
AND SCRIPT_TYPE='IMPLEMENTATION';
```

Example 18–6 Save IMPLEMENTATION Output in a Script File

The following statements save the IMPLEMENTATION output in a script file located at /myscript/mv_create2.sql:

```
CREATE DIRECTORY TUNE_RESULTS AS '/myscript'
GRANT READ, WRITE ON DIRECTRY TUNE_RESULTS TO PUBLIC;
EXECUTE DBMS_ADVISOR.CREATE_FILE(DBMS_ADVISOR.GET_TASK_SCRIPT('cust_mv2'),
   'TUNE_RESULTS', 'mv_create2.sql');
```

Fast Refreshable with Optimized Sub-Materialized View

The example illustrates how TUNE_MVIEW can optimize the materialized view so that fast refresh is possible. In the example, the materialized view's defining query with set operators is transformed into one sub-materialized view and one top-level materialized view. The subselect queries in the original defining query are of similar shape and their predicate expressions are combined.

The materialized view defining query contains a UNION set-operator so that the materialized view itself is not fast-refreshable. However, two subselect queries in the materialized view defining query can be combined as one single query.

Example 18–7 Optimized Sub-Materialized View for Fast Refresh

```
EXECUTE :task_cust_mv := 'cust_mv3';
EXECUTE :create_mv_ddl := '

CREATE MATERIALIZED VIEW cust_mv
REFRESH FAST ON DEMAND
ENABLE QUERY REWRITE AS
SELECT s.prod_id, s.cust_id, COUNT(*) cnt, SUM(s.amount_sold) sum_amount
FROM sales s, customers cs
WHERE s.cust_id = cs.cust_id AND s.cust_id IN (2005,1020)
GROUP BY s.prod_id, s.cust_id UNION
SELECT s.prod_id, s.cust_id, COUNT(*) cnt, SUM(s.amount_sold) sum_amount
FROM sales s, customers cs -
WHERE s.cust_id = cs.cust_id AND s.cust_id IN (1005,1010,1012)
GROUP BY s.prod_id, s.cust_id';

EXECUTE DBMS_ADVISOR.TUNE_MVIEW(:task_cust_mv, :create_mv_ddl);
```

The following recommendation will be made by TUNE_MVIEW with an optimized submaterialized view combining the two subselect queries and the submaterialized view is referenced by a new top-level materialized view as follows:

```
CREATE MATERIALIZED VIEW LOG ON "SH"."SALES"
  WITH ROWID, SEQUENCE ("PROD_ID","CUST_ID","AMOUNT_SOLD")
  INCLUDING NEW VALUES

ALTER MATERIALIZED VIEW LOG FORCE ON "SH"."SALES"
  ADD ROWID, SEQUENCE ("PROD_ID","CUST_ID","AMOUNT_SOLD")
  INCLUDING NEW VALUES

CREATE MATERIALIZED VIEW LOG ON "SH"."CUSTOMERS"
  WITH ROWID, SEQUENCE ("CUST_ID")  INCLUDING NEW VALUES

ALTER MATERIALIZED VIEW LOG FORCE ON "SH"."CUSTOMERS"
  ADD ROWID, SEQUENCE ("CUST_ID")  INCLUDING NEW VALUES

CREATE MATERIALIZED VIEW SH.CUST_MV$SUB1
  REFRESH FAST WITH ROWID
  ENABLE QUERY REWRITE AS
  SELECT SH.SALES.CUST_ID C1, SH.SALES.PROD_ID C2,
    SUM("SH"."SALES"."AMOUNT_SOLD") M1,
    COUNT("SH"."SALES"."AMOUNT_SOLD")M2, COUNT(*) M3
    FROM SH.CUSTOMERS, SH.SALES
    WHERE SH.SALES.CUST_ID = SH.CUSTOMERS.CUST_ID AND
    (SH.SALES.CUST_ID IN (2005, 1020, 1012, 1010, 1005))
    GROUP BY SH.SALES.CUST_ID, SH.SALES.PROD_ID

CREATE MATERIALIZED VIEW SH.CUST_MV
  REFRESH FORCE WITH ROWID ENABLE QUERY REWRITE AS
  (SELECT "CUST_MV$SUB1"."C2" "PROD_ID","CUST_MV$SUB1"."C1" "CUST_ID",
    "CUST_MV$SUB1"."M3" "CNT","CUST_MV$SUB1"."M1" "SUM_AMOUNT"
    FROM "SH"."CUST_MV$SUB1" "CUST_MV$SUB1"
    WHERE "CUST_MV$SUB1"."C1"=2005 OR "CUST_MV$SUB1"."C1"=1020)
    UNION
    (SELECT "CUST_MV$SUB1"."C2" "PROD_ID","CUST_MV$SUB1"."C1" "CUST_ID",
      "CUST_MV$SUB1"."M3" "CNT","CUST_MV$SUB1"."M1" "SUM_AMOUNT"
      FROM "SH"."CUST_MV$SUB1" "CUST_MV$SUB1"
      WHERE "CUST_MV$SUB1"."C1"=1012 OR "CUST_MV$SUB1"."C1"=1010 OR
            "CUST_MV$SUB1"."C1"=1005)
```

```
DBMS_ADVANCED_REWRITE.BUILD_SAFE_REWRITE_EQUIVALENCE ('SH.CUST_MV$RWEQ',
      'SELECT s.prod_id, s.cust_id, COUNT(*) cnt,
       SUM(s.amount_sold) sum_amount
       FROM sales s, customers cs
       WHERE s.cust_id = cs.cust_id AND s.cust_id IN (2005,1020)
       GROUP BY s.prod_id, s.cust_id UNION
       SELECT s.prod_id, s.cust_id, COUNT(*) cnt,
       SUM(s.amount_sold) sum_amount
       FROM sales s, customers cs
       WHERE s.cust_id = cs.cust_id AND s.cust_id IN (1005,1010,1012)
       GROUP BY s.prod_id, s.cust_id',
      '(SELECT "CUST_MV$SUB1"."C2" "PROD_ID",
        "CUST_MV$SUB1"."C1" "CUST_ID",
        "CUST_MV$SUB1"."M3" "CNT","CUST_MV$SUB1"."M1" "SUM_AMOUNT"
        FROM "SH"."CUST_MV$SUB1" "CUST_MV$SUB1"
        WHERE "CUST_MV$SUB1"."C1"=2005OR "CUST_MV$SUB1"."C1"=1020)
       UNION
       (SELECT "CUST_MV$SUB1"."C2" "PROD_ID",
        "CUST_MV$SUB1"."C1" "CUST_ID",
        "CUST_MV$SUB1"."M3" "CNT","CUST_MV$SUB1"."M1" "SUM_AMOUNT"
        FROM "SH"."CUST_MV$SUB1" "CUST_MV$SUB1"
        WHERE "CUST_MV$SUB1"."C1"=1012 OR "CUST_MV$SUB1"."C1"=1010 OR
              "CUST_MV$SUB1"."C1"=1005)',
      1811223110);
```

The original defining query of `cust_mv` has been optimized by combining the predicate of the two subselect queries in the sub-materialized view `CUST_MV$SUB1`. The required materialized view logs are also added to enable fast refresh of the submaterialized views.

The `DROP` output is as follows:

```
DROP MATERIALIZED VIEW SH.CUST_MV$SUB1
DROP MATERIALIZED VIEW SH.CUST_MV
DBMS_ADVANCED_REWRITE.DROP_REWRITE_EQUIVALENCE('SH.CUST_MV$RWEQ');
```

The following statements save the `IMPLEMENTATION` output in a script file located at `/myscript/mv_create3.sql`:

```
CREATE DIRECTORY TUNE_RESULTS AS '/myscript'
GRANT READ, WRITE ON DIRECTORY TUNE_RESULTS TO PUBLIC;
EXECUTE DBMS_ADVISOR.CREATE_FILE(DBMS_ADVISOR.GET_TASK_SCRIPT('cust_mv3'),
   'TUNE_RESULTS', 'mv_create3.sql');
```

19

Using Optimizer Hints

Optimizer hints can be used with SQL statements to alter execution plans. This chapter explains how to use hints to instruct the optimizer to use specific approaches.

The chapter contains the following sections:

Understanding Optimizer Hints

Hints let you make decisions usually made by the optimizer. As an application designer, you might know information about your data that the optimizer does not know. Hints provide a mechanism to instruct the optimizer to choose a certain query execution plan based on the specific criteria.

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 instruct the optimizer to use the optimal execution plan.

Note: The use of hints involves extra code that must be managed, checked, and controlled.

Types of Hints

Hints can be of the following general types:

- Single-table

  Single-table hints are specified on one table or view. `INDEX` and `USE_NL` are examples of single-table hints.

- Multi-table

  Multi-table hints are like single-table hints, except that the hint can specify one or more tables or views. `LEADING` is an example of a multi-table hint. Note that `USE_NL(table1 table2)` is not considered a multi-table hint because it is actually a shortcut for `USE_NL(table1)` and `USE_NL(table2)`.

- Query block

  Query block hints operate on single query blocks. `STAR_TRANSFORMATION` and `UNNEST` are examples of query block hints.

- Statement

  Statement hints apply to the entire SQL statement. ALL_ROWS is an example of a statement hint.

Hints by Category

Optimizer hints are grouped into the following categories:

- Hints for Optimization Approaches and Goals
- Hints for Access Paths
- Hints for Query Transformations
- Hints for Join Orders
- Hints for Join Operations
- Hints for Parallel Execution
- Additional Hints

These categories, and the hints contained within each category, are listed in the sections that follow.

> **See Also:** *Oracle Database SQL Reference* for syntax and a more detailed description of each hint

Hints for Optimization Approaches and Goals

The following hints let you choose between optimization approaches and goals:

- ALL_ROWS
- FIRST_ROWS(*n*)

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 specify 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.

If you specify either the ALL_ROWS or the FIRST_ROWS(*n*) 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 to subsequently choose an execution plan. These estimates might not be as accurate as those gathered by the DBMS_STATS package, so you should 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(*n*) hint, then the optimizer gives precedence to the access paths and join operations specified by the hints.

See "Optimization Approaches and Goal Hints in Views" on page 19-10 for hint behavior with mergeable views.

Hints for Access Paths

Each of the following hints instructs the optimizer to use a specific access path for a table:

- FULL
- CLUSTER
- HASH
- INDEX
- NO_INDEX
- INDEX_ASC
- INDEX_COMBINE
- INDEX_JOIN
- INDEX_DESC
- INDEX_FFS
- NO_INDEX_FFS
- INDEX_SS
- INDEX_SS_ASC
- INDEX_SS_DESC
- NO_INDEX_SS

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.

See "Access Path and Join Hints on Views" on page 19-10 and "Access Path and Join Hints Inside Views" on page 19-10 for hint behavior with mergeable views.

Note: For access path hints, Oracle ignores the hint if you specify the `SAMPLE` option in the `FROM` clause of a `SELECT` statement.

See Also: *Oracle Database SQL Reference* for more information on the `SAMPLE` option

Hints for Query Transformations

Each of the following hints instructs the optimizer to use a specific SQL query transformation:

- NO_QUERY_TRANSFORMATION
- USE_CONCAT
- NO_EXPAND
- REWRITE

- NO_REWRITE
- MERGE
- NO_MERGE
- STAR_TRANSFORMATION
- NO_STAR_TRANSFORMATION
- FACT
- NO_FACT
- UNNEST
- NO_UNNEST

Hints for Join Orders

The following hints suggest join orders:

- LEADING
- ORDERED

Hints for Join Operations

Each of the following hints instructs the optimizer to use a specific join operation for a table:

- USE_NL
- NO_USE_NL
- USE_NL_WITH_INDEX
- USE_MERGE
- NO_USE_MERGE
- USE_HASH
- NO_USE_HASH

Use of the `USE_NL` and `USE_MERGE` hints is recommended with any join order hint. See "Hints for Join Orders" on page 19-4. Oracle uses these hints when the referenced table is forced to be the inner table of a join; the hints are ignored if the referenced table is the outer table.

See "Access Path and Join Hints on Views" on page 19-10 and "Access Path and Join Hints Inside Views" on page 19-10 for hint behavior with mergeable views.

Hints for Parallel Execution

The hints that follow instruct the optimizer about how statements are parallelized or not parallelized when using parallel execution:

- PARALLEL
- PQ_DISTRIBUTE
- PARALLEL_INDEX
- NO_PARALLEL_INDEX

See "Parallel Execution Hints on Views" on page 19-10 and "Parallel Execution Hints Inside Views" on page 19-10 for hint behavior with mergeable views.

See Also: *Oracle Database Data Warehousing Guide* for more information on parallel execution

Additional Hints

The following are several additional hints:

- APPEND
- NOAPPEND
- CACHE
- NOCACHE
- PUSH_PRED
- NO_PUSH_PRED
- PUSH_SUBQ
- NO_PUSH_SUBQ
- QB_NAME
- CURSOR_SHARING_EXACT
- DRIVING_SITE
- DYNAMIC_SAMPLING
- MODEL_MIN_ANALYSIS

Specifying Hints

Hints apply only to the optimization of the block of a statement 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 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.

The following sections discuss the use of hints in more detail.

- Specifying a Full Set of Hints
- Specifying a Query Block in a Hint
- Specifying Global Table Hints
- Specifying Complex Index Hints

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 Example 19–1, the LEADING hint specifies the exact join order to be used; the join methods to be used on the different tables are also specified.

Example 19–1 Specifying a Full Set of Hints

```
SELECT /*+ LEADING(e2 e1) USE_NL(e1) INDEX(e1 emp_emp_id_pk)
           USE_MERGE(j) FULL(j) */
    e1.first_name, e1.last_name, j.job_id, sum(e2.salary) total_sal
  FROM employees e1, employees e2, job_history j
  WHERE e1.employee_id = e2.manager_id
    AND e1.employee_id = j.employee_id
    AND e1.hire_date = j.start_date
  GROUP BY e1.first_name, e1.last_name, j.job_id
  ORDER BY total_sal;
```

Specifying a Query Block in a Hint

To identify a query block in a query, an optional query block name can be used in a hint to specify the query block to which the hint applies. The syntax of the query block argument is of the form @*queryblock*, where *queryblock* is an identifier that specifies a query block in the query. The *queryblock* identifier can either be system-generated or user-specified.

- The system-generated identifier can be obtained by using EXPLAIN PLAN for the query. Pre-transformation query block names can be determined by running EXPLAIN PLAN for the query using the NO_QUERY_TRANSFORMATION hint.
- The user-specified name can be set with the QB_NAME hint.

In Example 19–2, the query block name is used with the NO_UNNEST hint to specify a query block in a SELECT statement on the view.

Example 19–2 Using a Query Block in a Hint

```
CREATE OR REPLACE VIEW v AS
  SELECT e1.first_name, e1.last_name, j.job_id, sum(e2.salary) total_sal
    FROM employees e1,
      ( SELECT *
          FROM employees e3) e2, job_history j
         WHERE e1.employee_id = e2.manager_id
          AND e1.employee_id = j.employee_id
          AND e1.hire_date = j.start_date
          AND e1.salary = ( SELECT max(e2.salary)
                            FROM employees e2
                            WHERE e2.department_id = e1.department_id )
  GROUP BY e1.first_name, e1.last_name, j.job_id
  ORDER BY total_sal;
```

After running EXPLAIN PLAN for the query and displaying the plan table output, you can determine the system-generated query block identifier. For example, a query block name is displayed in the following plan table output:

```
SELECT PLAN_TABLE_OUTPUT FROM TABLE(DBMS_XPLAN.DISPLAY(NULL, NULL, 'SERIAL'));
...
Query Block Name / Object Alias (identified by operation id):
-------------------------------------------------------------
...
```

```
  10 - SEL$4         / E2@SEL$4
```

After the query block name is determined it can be used in the following SQL statement:

```
SELECT /*+ NO_UNNEST( @SEL$4 ) */
  *
 FROM v;
```

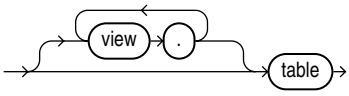
Specifying Global Table Hints

Hints that specify a table generally refer to tables in the `DELETE`, `SELECT`, or `UPDATE` query block 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. Table hints described in this chapter can be transformed into a global hint by using an extended `tablespec` syntax that includes view names with the table name.

In addition, an optional query block name can precede the `tablespec` syntax. See "Specifying a Query Block in a Hint" on page 19-6.

Hints that specify a table use the following syntax:

tablespec::=



where:

- `view` specifies a view name
- `table` specifies the name or alias of the table

If the view path is specified, the hint is resolved from left to right, where the first view must be present in the `FROM` clause, and each subsequent view must be specified in the `FROM` clause of the preceding view.

For example, in Example 19–3 a view `v` is created to return the first and last name of the employee, his or her first job and the total salary of all direct reports of that employee for each employee with the highest salary in his or her department. When querying the data, you want to force the use of the index `emp_job_ix` for the table `e3` in view `e2`.

Example 19–3 Using Global Hints Example

```
CREATE OR REPLACE VIEW v AS
SELECT
    e1.first_name, e1.last_name, j.job_id, sum(e2.salary) total_sal
FROM employees e1,
        ( SELECT *
          FROM employees e3) e2, job_history j
WHERE e1.employee_id = e2.manager_id
     AND e1.employee_id = j.employee_id
     AND e1.hire_date = j.start_date
     AND e1.salary = ( SELECT
                       max(e2.salary)
                       FROM employees e2
                       WHERE e2.department_id = e1.department_id)
GROUP BY e1.first_name, e1.last_name, j.job_id
ORDER BY total_sal;
```

By using the global hint structure, you can avoid the modification of view v with the specification of the index hint in the body of view e2. To force the use of the index emp_job_ix for the table e3, you can use one of the following:

```
SELECT /*+ INDEX(v.e2.e3 emp_job_ix) */  *
  FROM v;

SELECT /*+ INDEX(@SEL$2 e2.e3 emp_job_ix) */ *
  FROM v;

SELECT /*+ INDEX(@SEL$3 e3 emp_job_ix) */ *
  FROM v;
```

The global hint syntax also applies to unmergeable views as in Example 19–4.

Example 19–4 Using Global Hints with NO_MERGE

```
CREATE OR REPLACE VIEW v1 AS
  SELECT *
    FROM employees
    WHERE employee_id < 150;

CREATE OR REPLACE VIEW v2 AS
  SELECT v1.employee_id employee_id, departments.department_id department_id
    FROM v1, departments
    WHERE v1.department_id = departments.department_id;

SELECT /*+ NO_MERGE(v2) INDEX(v2.v1.employees emp_emp_id_pk)
                        FULL(v2.departments) */ *
  FROM v2
  WHERE department_id = 30;
```

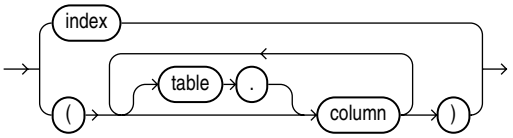
The hints cause v2 not to be merged and specify access path hints for the employee and department tables. These hints are pushed down into the (nonmerged) view v2.

See Also: "Using Hints with Views" on page 19-9

Specifying Complex Index Hints

Hints that specify an index can use either a simple index name or a parenthesized list of columns as follows:

indexspec::=



where:

- table specifies the name
- column specifies the name of a column in the specified table
  - The columns can optionally be prefixed with table qualifiers allowing the hint to specify bitmap join indexes where the index columns are on a different table than the indexed table. If tables qualifiers are present, they must be base tables, not aliases in the query.

  - Each column in an index specification must be a base column in the specified table, not an expression. Function-based indexes cannot be hinted using a column specification unless the columns specified in the index specification form the prefix of a function-based index.

- `index` specifies an index name

  When `tablespec` is followed by `indexspec` in the specification of a hint, a comma separating the table name and index name is permitted but not required. Commas are also permitted, but not required, to separate multiple occurrences of `indexspec`.

The hint is resolved as follows:

- If an index name is specified, only that index is considered.
- If a column list is specified and an index exists whose columns match the specified columns in number and order, only that index is considered. If no such index exists, then any index on the table with the specified columns as the prefix in the order specified is considered. In either case, the behavior is exactly as if the user had specified the same hint individually on all the matching indexes.

For example, in Example 19–3 the `job_history` table has a single-column index on the `employee_id` column and a concatenated index on `employee_id` and `start_date` columns. To specifically instruct the optimizer on index use, the query can be hinted as follows:

```
SELECT /*+ INDEX(v.j jhist_employee_ix (employee_id start_date)) */ * FROM v;
```

Using Hints with Views

Oracle does not encourage the use of hints inside or on views (or subqueries). This is because you can define views in one context and use them in another. Also, such hints can result in unexpected execution 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 want to specify a hint for a table in a view or subquery, then the global hint syntax is recommended. See "Specifying Global Table Hints" on page 19-7.

If you decide, nonetheless, to use hints with views, the following sections describe the behavior in each case.

- Hints and Complex Views
- Hints and Mergeable Views
- Hints and Nonmergeable Views

Hints and Complex 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.

Hints and Mergeable Views

This section describes hint behavior with mergeable views.

Optimization Approaches and Goal Hints in Views

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 an inline view (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 non-inline views, access path and join hints in the view are preserved only if the referencing 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`, `NO_PARALLEL`, `PARALLEL_INDEX`, and `NO_PARALLEL_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`, `NO_PARALLEL`, `PARALLEL_INDEX`, and `NO_PARALLEL_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.

20

Using Plan Stability

This chapter describes how to use plan stability to preserve performance characteristics. Plan stability also facilitates migration from the rule-based optimizer to the query optimizer when you upgrade to a new Oracle release.

This chapter contains the following topics:

- Using Plan Stability to Preserve Execution Plans
- Using Plan Stability with Query Optimizer Upgrades

> **Note:** Stored outlines will be desupported in a future release in favor of SQL plan management. In Oracle Database 11*g* Release 1 (11.1), stored outlines continue to function as in past releases. However, Oracle strongly recommends that you use SQL plan management for new applications. SQL plan management creates SQL plan baselines, which offer superior SQL performance and stability compared with stored outlines.
>
> If you have existing stored outlines, consider migrating them to SQL plan baselines by using the `LOAD_PLANS_FROM_CURSOR_CACHE` or `LOAD_PLANS_FROM_SQLSET` procedure of the `DBMS_SPM` package. When the migration is complete, you should disable or remove the stored outlines.

See Also:

- Chapter 15, "Using SQL Plan Management" for information about SQL plan management
- *Oracle Database PL/SQL Packages and Types Reference* for information about the `DBMS_SPM` package

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. An outline is implemented as a set of optimizer hints that are associated with the SQL statement. If the use of the

outline is enabled for the statement, Oracle automatically considers the stored hints and tries to generate an execution plan in accordance with those hints.

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.

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 the Oracle 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 allow 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`, and the parameter must also set to `SIMILAR` when attempting to use the outline. See Chapter 7, "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 query optimization. Query 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.

Some of the useful DBMS_OUTLN and DBMS_OUTLN_EDIT procedures are:

- CLEAR_USED - Clears specified outline
- DROP_BY_CAT - Drops outlines that belong to a specified category
- UPDATE_BY_CAT - Changes the category of outlines in one specified category to a new specified category
- EXACT_TEXT_SIGNATURES - Computes an outline signature according to an exact text matching scheme
- GENERATE_SIGNATURE - Generates a signature for the specified SQL text

See Also:

- *Oracle Database PL/SQL Packages and Types Reference* for detailed information on using DBMS_OUTLN package procedures
- *Oracle Database PL/SQL Packages and Types Reference* for detailed information on using DBMS_OUTLN_EDIT package 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 initialization 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.

When creating or editing a private outline, the outline data is written to global temporary tables in the SYSTEM schema. These tables are accessible with the OL$, OL$HINTS, and OL$NODES synonyms.

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 initialization 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 initialization 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.

See Also:

- *Oracle Database SQL Reference* for more information on the CREATE OUTLINE statement
- *Oracle Database PL/SQL Packages and Types Reference* for more information on the DBMS_OUTLN and DBMS_OUTLN_EDIT packages
- "Moving from RBO to the Query Optimizer" on page 20-9 for information on moving from the rule-based optimizer to the query optimizer
- *Oracle Enterprise Manager Concepts* for information on 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 initialization 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` initialization 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`.

Using and Editing Stored Outlines

When you activate the use of stored outlines, Oracle always uses the query optimizer. This is because outlines rely on hints, and to be effective, most hints require the query 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.

If you want to use a specific outline rather than all the outlines in a category, use the `ALTER OUTLINE` statement to enable the specific outline. If you want to use the outlines in a category except for a specific outline, use the `ALTER OUTLINE` statement to disable the specific outline in the category that is being used. The `ALTER OUTLINE` statement can also rename a stored outline, reassign it to a different category, or regenerate it.

> **See Also:** *Oracle Database SQL Reference* for information on the `ALTER OUTLINE` statement

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 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 finished editing, this parameter should be set to false to restore the session to normal outline lookup according to the USE_STORED_OUTLINES parameter.

Note: The 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 *Oracle Database SQL Reference.*

You also can use the Oracle Enterprise Manager Outline Editor to update outlines.

> **See Also:** *Oracle Enterprise Manager Concepts* for information on Oracle Enterprise Manager GUI tools

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. Clone the outline being edited to the private area using the following:

   ```
   CREATE PRIVATE OUTLINE p_ol1 FROM ol1;
   ```

3. Edit the outline, either with the Outline Editor in Enterprise Manager or manually using DBMS_OUTLN_EDIT. If you want to change join order, modify the appropriate LEADING hint. See "Hints for Join Orders" on page 19-4.
4. If manually editing the outline, use DBMS_OUTLN_EDIT.CHANGE_JOIN_POS to change the position; then resynchronize the stored outline definition using either of the following so-called identity statements:

   ```
   exec DBMS_OUTLN_EDIT.REFRESH_PRIVATE_OUTLINE ('S-I1');

   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.
5. Test the edits. Set USE_PRIVATE_OUTLINES=TRUE, and issue the outline statement or run EXPLAIN PLAN on the statement.
6. 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;
   ```

7. Disable private outline usage by setting the following:

```
USE_PRIVATE_OUTLINES=FALSE
```

See Also:

- *Oracle Database Reference* for syntax when using the CREATE_STORED_OUTLINES initialization parameter
- *Oracle Database SQL Reference* for SQL syntax when using the USE_STORED_OUTLINES and USE_PRIVATE_OUTLINES parameters
- *Oracle Database PL/SQL Packages and Types 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';
```

You can check the flags in _OUTLINES views for information on compatibility, format, and whether an outline is enabled. For example, check the ENABLED field in the USER_OUTLINES view to determine whether an outline is enabled or not.

```
SELECT NAME, CATEGORY, ENABLED FROM USER_OUTLINES;
```

> **See Also:** *Oracle Database Reference* for information on views related to outlines

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 SYSTEM tablespace using a schema called OUTLN. If outlines use too much space in the SYSTEM tablespace, then you can move them. To do this, create a separate tablespace and move the outline tables into it using the following process.

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.

1. Use the Oracle Export utility to export the OL$, OL$HINTS, and OL$NODES tables:

```
EXP OUTLN/outln_password
    FILE = exp_file TABLES = 'OL$' 'OL$HINTS' 'OL$NODES'
```

2. Start SQL*Plus and connect to the database.

```
CONNECT OUTLN/outln_password;
```

3. Remove the previous OL$, OL$HINTS, and OL$NODES tables:

```
DROP TABLE OL$;
DROP TABLE OL$HINTS;
DROP TABLE OL$NODES;
```

4. Create a new tablespace for the tables:

```
CONNECT SYSTEM/system_password;
CREATE TABLESPACE outln_ts
  DATAFILE 'tspace.dat' SIZE 2M
  DEFAULT STORAGE (INITIAL 10K NEXT 20K MINEXTENTS 1 MAXEXTENTS 999
                   PCTINCREASE 10)
  ONLINE;
```

5. Enter the following statement to change the default tablespace:

```
ALTER USER OUTLN DEFAULT TABLESPACE outln_ts;
```

6. To force the import into the OUTLN_TS tablespace, set quota for the SYSTEM tablespace to 0K for the OUTLN user. You will also need to revoke the UNLIMITED TABLESPACE privilege and all roles, such as the RESOURCE role, that have unlimited tablespace privileges or quotas. Set a quota for the OUTLN tablespace.
7. Import the OL$, OL$HINTS, and OL$NODES tables:

```
IMP OUTLN/outln_password
    FILE = exp_file TABLES = (OL$, OL$HINTS, OL$NODES)
```

When the import process has finished, the OL$, OL$HINTS, and OL$NODES tables are re-created in the schema named OUTLN and now reside in a new tablespace called OUTLN_TS.

At the completion of the process, you may want to adjust the tablespace quotas for the OUTLN user appropriately by adding any privileges and roles that were removed in a previous step.

See Also:

- *Oracle Database Utilities* for detailed information on using the EXPORT and IMPORT utilities, note the section on reorganizing tablespaces under the discussion of the IMPORT utility
- *Oracle Database PL/SQL Packages and Types Reference* for detailed information on using the DBMS_OUTLN package

Using Plan Stability with Query Optimizer Upgrades

This section describes procedures you can use to significantly improve performance by taking advantage of query optimizer functionality. Plan stability provides a way to preserve a system's targeted execution plans with satisfactory performance while also taking advantage of new query optimizer features for the rest of the SQL statements.

While there are classes of SQL statements and features where an exact reproduction of the original execution plan is not guaranteed, plan stability can still be a highly useful part of the migration process. Before the migration, outline capturing of execution plan should be turned on until all or most of the applications SQL-statement have been covered. If, after the migration, there are performance problems for some specific SQL-statement, the use of the stored outline for that statement can be turned on as a way of restoring the old behavior. The use of stored outlines is not always the best way of resolving a migration related performance problem because it prevents plans from adapting to changing data properties, but it adds to the arsenal of techniques that can be used to address such problems.

Topics covered in this section are:

- Moving from RBO to the Query Optimizer
- Moving to a New Oracle Release under the Query Optimizer

Moving from RBO to the Query 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 query optimization.

By creating outlines for an application before switching to query optimization, the plans generated by the rule-based optimizer can be used, while statements generated by newly written applications developed after the switch use query 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 query 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.

Moving to a New Oracle Release under the Query Optimizer

When upgrading to a new Oracle release under query 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"'
```

21

Using Application Tracing Tools

Oracle provides several tracing tools that can help you monitor and analyze applications running against an Oracle database.

End to End Application Tracing can identify the source of an excessive workload, such as a high load SQL statement, by client identifier, service, module, action, session, instance, or an entire database. This isolates the problem to a specific user, service, session, or application component.

Oracle provides the `trcsess` command-line utility that consolidates tracing information based on specific criteria.

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

This chapter contains the following sections:

> **See Also:** *SQL*Plus User's Guide and Reference* for information about the use of Autotrace to trace and tune SQL*Plus statements

End to End Application Tracing

End to End Application Tracing simplifies the process of diagnosing performance problems in a multitier environments. In multitier environments, a request from an end client is routed to different database sessions by the middle tier making it difficult to track a client across different database sessions. End to End Application Tracing uses a client identifier to uniquely trace a specific end-client through all tiers to the database server.

This feature could identify the source of an excessive workload, such as a high load SQL statement, and allow you to contact the specific user responsible. Also, a user having problems can contact you and then you can identify what that user's session is doing at the database level.

End to End Application Tracing also simplifies management of application workloads by tracking specific modules and actions in a service.

Workload problems can be identified by End to End Application Tracing for:

- Client identifier - specifies an end user based on the logon Id, such as `HR.HR`
- Service - specifies a group of applications with common attributes, service level thresholds, and priorities; or a single application, such as `ACCTG` for an accounting application
- Module - specifies a functional block, such as Accounts Receivable or General Ledger, of an application
- Action - specifies an action, such as an INSERT or UPDATE operation, in a module
- Session - specifies a session based on a given database session identifier (SID), on the local instance
- Instance - specifies a given instance based on the instance name

After tracing information is written to files, the information can be consolidated by the `trcsess` utility and diagnosed with an analysis utility such as `TKPROF`.

To to create services on single instance Oracle databases, you can use the `CREATE_SERVICE` procedure in the `DBMS_SERVICE` package or set the `SERVICE_NAMES` initialization parameter.

The module and action names are set by the application developer. For example, you would use the `SET_MODULE` and `SET_ACTION` procedures in the `DBMS_APPICATION_INFO` package to set these values in a PL/SQL program.

See Also:

- *Oracle Database Concepts* for information on services
- *Oracle Call Interface Programmer's Guide* for information on setting the necessary parameters in an OCI application
- *Oracle Database PL/SQL Packages and Types Reference* for information on the `DBMS_MONITOR`, `DBMS_SESSION`, `DBMS_SERVICE`, and `DBMS_APPICATION_INFO` packages
- *Oracle Database Reference* for information on V$ views and initialization parameters

Accessing End to End Tracing with Oracle Enterprise Manager

The primary interface for End to End Application Tracing is the Oracle Enterprise Manager.

To manage End to End Application Tracing using Oracle Enterprise Manager:

1. On the Database Performance page, under Additional Monitoring Links, click **Top Consumers**.

   The Top Consumers page appears.

2. Click the **Top Services**, **Top Modules**, **Top Actions**, **Top Clients**, or **Top Sessions** links to display the top consumers for each consumer type.
3. On the individual Top Consumers page for each consumer type, enable or disable statistics gathering and SQL tracing for specific consumers.

Managing End to End Tracing with APIs and Views

While the primary interface for End to End Application Tracing is the Oracle Enterprise Manager Database Control, this feature can be managed with DBMS_MONITOR package APIs.

Enabling and Disabling Statistic Gathering for End to End Tracing

To gather the appropriate statistics using PL/SQL, you need to enable statistics gathering for client identifier, service, module, or action using procedures in the DBMS_MONITOR package.

You can gather statistics by the following criteria:

- Statistic Gathering for Client Identifier
- Statistic Gathering for Service, Module, and Action

The default level is the session-level statistics gathering. Statistics gathering is global for the database and continues after an instance is restarted.

Statistic Gathering for Client Identifier The procedure CLIENT_ID_STAT_ENABLE enables statistic gathering for a given client identifier. For example, to enable statistics gathering for a specific client identifier:

```
EXECUTE DBMS_MONITOR.CLIENT_ID_STAT_ENABLE(client_id => 'OE.OE');
```

In the example, OE.OE is the client identifier for which you want to collect statistics. You can view client identifiers in the CLIENT_IDENTIFIER column in V$SESSION.

The procedure CLIENT_ID_STAT_DISABLE disables statistic gathering for a given client identifier. For example:

```
EXECUTE DBMS_MONITOR.CLIENT_ID_STAT_DISABLE(client_id => 'OE.OE');
```

Statistic Gathering for Service, Module, and Action The procedure SERV_MOD_ACT_STAT_ENABLE enables statistic gathering for a combination of service, module, and action. For example:

```
EXECUTE DBMS_MONITOR.SERV_MOD_ACT_STAT_ENABLE(service_name => 'ACCTG',
        module_name => 'PAYROLL');

EXECUTE DBMS_MONITOR.SERV_MOD_ACT_STAT_ENABLE(service_name => 'ACCTG',
        module_name => 'GLEDGER', action_name => 'INSERT ITEM');
```

If both of the previous commands are executed, statistics are gathered as follows:

- For the ACCTG service, because accumulation for each service name is the default
- For all actions in the PAYROLL module
- For the INSERT ITEM action within the GLEDGER module

The procedure SERV_MOD_ACT_STAT_DISABLE disables statistic gathering for a combination of service, module, and action. For example:

```
EXECUTE DBMS_MONITOR.SERV_MOD_ACT_STAT_DISABLE(service_name => 'ACCTG',
        module_name => 'GLEDGER', action_name => 'INSERT ITEM');
```

Viewing Gathered Statistics for End to End Application Tracing

The statistics that have been gathered can be displayed with a number of dynamic views.

- The accumulated global statistics for the currently enabled statistics can be displayed with the DBA_ENABLED_AGGREGATIONS view.
- The accumulated statistics for a specified client identifier can be displayed in the V$CLIENT_STATS view.
- The accumulated statistics for a specified service can be displayed in V$SERVICE_STATS view.
- The accumulated statistics for a combination of specified service, module, and action can be displayed in the V$SERV_MOD_ACT_STATS view.
- The accumulated statistics for elapsed time of database calls and for CPU use can be displayed in the V$SERVICEMETRIC view.

Enabling and Disabling for End to End Tracing

To enable tracing for client identifier, service, module, action, session, instance or database, you need to execute the appropriate procedures in the DBMS_MONITOR package. You can enable tracing for specific diagnosis and workload management by the following criteria:

- Tracing for Client Identifier
- Tracing for Service, Module, and Action
- Tracing for Session
- Tracing for Entire Instance or Database

With the criteria that you provide, specific trace information is captured in a set of trace files and combined into a single output trace file.

Tracing for Client Identifier The CLIENT_ID_TRACE_ENABLE procedure enables tracing globally for the database for a given client identifier. For example:

```
EXECUTE DBMS_MONITOR.CLIENT_ID_TRACE_ENABLE(client_id => 'OE.OE',
        waits => TRUE, binds => FALSE);
```

In this example, OE.OE is the client identifier for which SQL tracing is to be enabled. The TRUE argument specifies that wait information will be present in the trace. The FALSE argument specifies that bind information will not be present in the trace.

The CLIENT_ID_TRACE_DISABLE procedure disables tracing globally for the database for a given client identifier. To disable tracing, for the previous example:

```
EXECUTE DBMS_MONITOR.CLIENT_ID_TRACE_DISABLE(client_id => 'OE.OE');
```

Tracing for Service, Module, and Action The SERV_MOD_ACT_TRACE_ENABLE procedure enables SQL tracing for a given combination of service name, module, and action globally for a database, unless an instance name is specified in the procedure.

```
EXECUTE DBMS_MONITOR.SERV_MOD_ACT_TRACE_ENABLE(service_name => 'ACCTG',
        waits => TRUE, binds => FALSE, instance_name => 'inst1');
```

In this example, the service ACCTG is specified. The module or action name is not specified. The TRUE argument specifies that wait information will be present in the trace. The FALSE argument specifies that bind information will not be present in the trace. The inst1 instance is specified to enable tracing only for that instance.

To enable tracing for all actions for a given combination of service and module:

```
EXECUTE DBMS_MONITOR.SERV_MOD_ACT_TRACE_ENABLE(service_name => 'ACCTG',
        module_name => 'PAYROLL', waits => TRUE,  binds => FALSE,
```

```
        instance_name => 'inst1');
```

The SERV_MOD_ACT_TRACE_DISABLE procedure disables the trace at all enabled instances for a given combination of service name, module, and action name globally. For example, the following disables tracing for the first example in this section:

```
EXECUTE DBMS_MONITOR.SERV_MOD_ACT_TRACE_DISABLE(service_name => 'ACCTG',
        instance_name => 'inst1');
```

This example disables tracing for the second example in this section:

```
EXECUTE DBMS_MONITOR.SERV_MOD_ACT_TRACE_DISABLE(service_name => 'ACCTG',
        module_name => 'PAYROLL', instance_name => 'inst1');
```

Tracing for Session The SESSION_TRACE_ENABLE procedure enables the trace for a given database session identifier (SID), on the local instance.

To enable tracing for a specific session ID and serial number, determine the values for the session that you want to trace:

```
SELECT SID, SERIAL#, USERNAME FROM V$SESSION;

       SID    SERIAL# USERNAME
---------- ---------- ------------------------------
        27         60 OE
...
```

Use the appropriate values to enable tracing for a specific session:

```
EXECUTE DBMS_MONITOR.SESSION_TRACE_ENABLE(session_id => 27, serial_num => 60,
        waits => TRUE, binds => FALSE);
```

The TRUE argument specifies that wait information will be present in the trace. The FALSE argument specifies that bind information will not be present in the trace.

The SESSION_TRACE_DISABLE procedure disables the trace for a given database session identifier (SID) and serial number. For example:

```
EXECUTE DBMS_MONITOR.SESSION_TRACE_DISABLE(session_id => 27, serial_num => 60);
```

While the DBMS_MONITOR package can only be invoked by a user with the DBA role, any user can also enable SQL tracing for their own session by using the DBMS_SESSION package. The SESSION_TRACE_ENABLE procedure can be invoked by any user to enable session-level SQL trace for their own session. For example:

```
EXECUTE DBMS_SESSION.SESSION_TRACE_ENABLE(waits => TRUE, binds => FALSE);
```

The TRUE argument specifies that wait information will be present in the trace. The FALSE argument specifies that bind information will not be present in the trace.

The SESSION_TRACE_DISABLE procedure disables the trace for the invoking session. For example:

```
EXECUTE DBMS_SESSION.SESSION_TRACE_DISABLE();
```

Tracing for Entire Instance or Database The DATABASE_TRACE_ENABLE procedure enables SQL tracing for a given instance or an entire database. For example:

```
EXECUTE DBMS_MONITOR.DATABASE_TRACE_ENABLE(waits => TRUE, binds => FALSE,
        instance_name => 'inst1');
```

In this example, the inst1 instance is specified to enable tracing for that instance. The TRUE argument specifies that wait information will be present in the trace. The FALSE

argument specifies that bind information will not be present in the trace. This example will result in SQL tracing of every SQL in the inst1 instance.

The DATABASE_TRACE_ENABLE procedure will override all other session-level traces, but will be complementary to the client identifier, service, module, and action traces. All new sessions will inherit the wait and bind information specified by this procedure until the DATABASE_TRACE_DISABLE procedure is called. When this procedure is invoked with the instance_name parameter specified, it will reset the session-level SQL trace for the named instance. If this procedure is invoked without the instance_name parameter specified, it will reset the session-level SQL trace for the entire database.

The DATABASE_TRACE_DISABLE procedure disables the tracing for an entire instance or database. For example:

```
EXECUTE DBMS_MONITOR.DATABASE_TRACE_DISABLE(instance_name => 'inst1');
```

In this example, all session-level SQL tracing will be disabled for the inst1 instance. To disable the session-level SQL tracing for an entire database, invoke the DATABASE_TRACE_DISABLE procedure without specifying the instance_name parameter:

```
EXECUTE DBMS_MONITOR.DATABASE_TRACE_DISABLE();
```

Viewing Enabled Traces for End to End Tracing

All outstanding traces can be displayed in an Oracle Enterprise Manager report or with the DBA_ENABLED_TRACES view. In the DBA_ENABLED_TRACES view, you can determine detailed information about how a trace was enabled, including the trace type. The trace type specifies whether the trace is enabled for client identifier, session, service, database, or a combination of service, module, and action.

Using the trcsess Utility

The trcsess utility consolidates trace output from selected trace files based on several criteria:

- Session Id
- Client Id
- Service name
- Action name
- Module name

After trcsess merges the trace information into a single output file, the output file could be processed by TKPROF.

trcsess is useful for consolidating the tracing of a particular session for performance or debugging purposes. Tracing a specific session is usually not a problem in the dedicated server model as a single dedicated process serves a session during its lifetime. All the trace information for the session can be seen from the trace file belonging to the dedicated server serving it. However, in a shared server configuration a user session is serviced by different processes from time to time. The trace pertaining to the user session is scattered across different trace files belonging to different processes. This makes it difficult to get a complete picture of the life cycle of a session.

Syntax for trcsess

The syntax for the trcsess utility is:

```
trcsess  [output=output_file_name]
         [session=session_id]
         [clientid=client_id]
         [service=service_name]
         [action=action_name]
         [module=module_name]
         [trace_files]
```

where

- `output` specifies the file where the output is generated. If this option is not specified, then standard output is used for the output.
- `session` consolidates the trace information for the session specified. The session identifier is a combination of session index and session serial number, such as `21.2371`. You can locate these values in the `V$SESSION` view.
- `clientid` consolidates the trace information given client Id.
- `service` consolidates the trace information for the given service name.
- `action` consolidates the trace information for the given action name.
- `module` consolidates the trace information for the given module name.
- *`trace_files`* is a list of all the trace file names, separated by spaces, in which `trcsess` should look for trace information. The wild card character `*` can be used to specify the trace file names. If trace files are not specified, all the files in the current directory are taken as input to `trcsess`.

One of the `session`, `clientid`, `service`, `action`, or `module` options must be specified. If more then one of the `session`, `clientid`, `service`, `action`, or `module` options is specified, then the trace files which satisfies all the criteria specified are consolidated into the output file.

Sample Output of trcsess

This sample output of `trcsess` shows the consolidation of traces for a particular session. In this example the session index and serial number is equal to `21.2371`.

`trcsess` can be invoked with various options. In the following case, all files in current directory are taken as input:

```
trcsess session=21.2371
```

In this case, several trace files are specified:

```
trcsess session=21.2371 main_12359.trc main_12995.trc
```

The sample output is similar to the following:

```
[PROCESS ID = 12359]
*** 2002-04-02 09:48:28.376
PARSING IN CURSOR #1 len=17 dep=0 uid=27 oct=3 lid=27 tim=868373970961
hv=887450622 ad='22683fb4'
select * from cat
END OF STMT
PARSE #1:c=0,e=339,p=0,cr=0,cu=0,mis=0,r=0,dep=0,og=4,tim=868373970944
EXEC #1:c=0,e=221,p=0,cr=0,cu=0,mis=0,r=0,dep=0,og=4,tim=868373971411
FETCH #1:c=0,e=791,p=0,cr=7,cu=0,mis=0,r=1,dep=0,og=4,tim=868373972435
FETCH #1:c=0,e=1486,p=0,cr=20,cu=0,mis=0,r=6,dep=0,og=4,tim=868373986238
*** 2002-04-02 10:03:58.058
XCTEND rlbk=0, rd_only=1
```

```
STAT #1 id=1 cnt=7 pid=0 pos=1 obj=0 op='FILTER  '
STAT #1 id=2 cnt=7 pid=1 pos=1 obj=18 op='TABLE ACCESS BY INDEX ROWID OBJ$ '
STAT #1 id=3 cnt=7 pid=2 pos=1 obj=37 op='INDEX RANGE SCAN I_OBJ2 '
STAT #1 id=4 cnt=0 pid=1 pos=2 obj=4 op='TABLE ACCESS CLUSTER TAB$J2 '
STAT #1 id=5 cnt=6 pid=4 pos=1 obj=3 op='INDEX UNIQUE SCAN I_OBJ# '
[PROCESS ID=12995]
*** 2002-04-02 10:04:32.738
Archiving is disabled
Archiving is disabled
```

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
- Wait event data for each SQL statement, and a summary for each trace file

If the cursor for the SQL statement is closed, SQL Trace also provides row source information that includes:

- Row operations showing the actual execution plan of each SQL statement
- Number of rows, number of consistent reads, number of physical reads, number of physical writes, and time elapsed for each operation on a row

Although it is possible to enable the SQL Trace facility for a session or for an instance, it is recommended that you use the DBMS_SESSION or DBMS_MONITOR packages instead. When the SQL Trace facility is enabled for a session or for an instance, performance statistics for all SQL statements executed in a user session or in the instance are placed into trace files. Using the SQL Trace facility can have a severe performance impact and may result in increased system overhead, excessive CPU usage, and inadequate disk space.

> **See Also:** "Enabling and Disabling for End to End Tracing" on page 21-4 for information on using the DBMS_SESSION or DBMS_MONITOR packages to enable SQL tracing for a session or an instance.

Oracle provides the trcsess command-line utility that consolidates tracing information from several trace files based on specific criteria, such as session or client Id. See "Using the trcsess Utility" on page 21-6.

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. TKPROF can also:

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

> **Note:** If the cursor for a SQL statement is not closed, TKPROF output does not automatically include the actual execution plan of the SQL statement. In this situation, you can use the EXPLAIN option with TKPROF to generate an execution plan.

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 21-9.

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 21-11.

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 21-12.

4. Interpret the output file created in Step 3.

   See "Step 4: Interpreting TKPROF Output" on page 21-15.

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 21-20.

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 the settings of the TIMED_STATISTICS, MAX_DUMP_FILE_SIZE, and USER_DUMP_DEST initialization parameters. See Table 21–1.

Table 21–1 Initialization Parameters to Check Before Enabling SQL Trace

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.

See Also:

- "Interpreting Statistics" on page 5-7 for considerations when setting the STATISTICS_LEVEL, DB_CACHE_ADVICE, TIMED_STATISTICS, or TIMED_OS_STATISTICS initialization parameters
- "Setting the Level of Statistics Collection" on page 10-7 for information about STATISTICS_LEVEL settings
- *Oracle Database Reference* for information on the STATISTICS_LEVEL initialization parameter
- *Oracle Database Reference* for information about the dynamic performance V$STATISTICS_LEVEL view

1. 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.

   You can also set the TRACEFILE_IDENTIFIER initialization parameter to specify a custom identifier that becomes part of the trace file name. For example, you can add *my_trace_id* to subsequent trace file names for easy identification with the following:

   ```
   ALTER SESSION SET TRACEFILE_IDENTIFIER = 'my_trace_id';
   ```

   See Also: *Oracle Database Reference* for information on the TRACEFILE_IDENTIFIER initialization parameter

2. 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.
3. 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.

See Also:

- "Setting the Level of Statistics Collection" on page 10-7 for information about `STATISTICS_LEVEL` settings
- *Oracle Database Reference* for information on the `STATISTICS_LEVEL` initialization parameter

Step 2: Enabling the SQL Trace Facility

Enable the SQL Trace facility for the session by using one of the following:

- `DBMS_SESSION.SET_SQL_TRACE` procedure
- `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. It is recommended that you use the DBMS_SESSION or DBMS_MONITOR packages to enable SQL tracing for a session or an instance instead. For more information on using these packages, see "Enabling and Disabling for End to End Tracing" on page 21-4.

You might need to modify an 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 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.

You can enable the SQL Trace facility for an instance by setting the value of the `SQL_TRACE` initialization parameter to `TRUE` in the initialization file.

```
SQL_TRACE = TRUE
```

After the instance has been restarted with the updated initialization parameter file, SQL Trace is enabled for the instance and statistics are collected for all sessions. If the SQL Trace facility has been enabled for the instance, you can disable it for the instance by setting the value of the `SQL_TRACE` parameter to `FALSE`.

Note: Setting `SQL_TRACE` to TRUE can have a severe performance impact. For more information, see *Oracle Database Reference*.

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.
- Run the trcsess command-line utility to consolidate tracing information from several trace files, then run TKPROF on the result. See "Using the trcsess Utility" on page 21-6.

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 is run from the operating system prompt. The syntax is:

```
tkprof filename1 filename2 [waits=yes|no] [sort=option] [print=n]
    [aggregate=yes|no] [insert=filename3] [sys=yes|no] [table=schema.table]
```

```
[explain=user/password] [record=filename4] [width=n]
```

The input and output files are the only required arguments. If you invoke TKPROF without arguments, then online help is displayed. Use the arguments in Table 21–2 with TKPROF.

Table 21–2 TKPROF Arguments

Argument	Description
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.
WAITS	Specifies whether to record summary for any wait events found in the trace file. Values are YES or NO. The default is YES.
SORTS	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.
USERID	Userid of user that parsed the cursor.

Table 21–2 (Cont.) TKPROF Arguments

Argument	Description
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.
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 *Oracle Database 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$PLAN_TABLE in the schema of the user specified by the EXPLAIN parameter. If you use the TABLE parameter without the EXPLAIN parameter, then TKPROF ignores the TABLE parameter. If no plan table exists, TKPROF creates the table PROF$PLAN_TABLE and then drops it at the end.
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 *filename4* with all of the nonrecursive SQL in the trace file. This can be used to replay the user events from the trace file.
WIDTH	An integer that controls the output line width of some TKPROF output, such as the explain plan. This parameter is useful for post-processing of TKPROF output.

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 21-24.

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.

  Note: If the cursor for a SQL statement is not closed, TKPROF output does not automatically include the actual execution plan of the SQL statement. In this situation, you can use the EXPLAIN option with TKPROF to generate an execution plan.

- 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
- Row Source Operations
- Wait Event Information
- Interpreting the Resolution of Statistics
- Understanding Recursive Calls
- 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. See Table 21–3.

Table 21–3 CALL Column Values

CALL Value	Meaning
PARSE	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	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	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, also called Logical I/Os (LIOs). See Table 21–4.

Table 21–4 SQL Trace Statistics for Parses, Executes, and Fetches.

SQL Trace Statistic	Meaning
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. See Table 21–5.

Table 21–5 SQL Trace Statistics for the ROWS Column

SQL Trace Statistic	Meaning
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; therefore, 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.

Row Source Operations

Row source operations provide the number of rows processed for each operation executed on the rows and additional row source information, such as physical reads and writes. The following is a sample:

```
Rows     Row Source Operation
-------  ---------------------------------------------------
      0  DELETE  (cr=43141 r=266947 w=25854 time=60235565 us)
  28144   HASH JOIN ANTI (cr=43057 r=262332 w=25854 time=48830056 us)
  51427    TABLE ACCESS FULL STATS$SQLTEXT (cr=3465 r=3463 w=0 time=865083 us)
 647529    INDEX FAST FULL SCAN STATS$SQL_SUMMARY_PK
                    (cr=39592 r=39325 w=0 time=10522877 us) (object id 7409)
```

In this sample TKPROF output, note the following under the Row Source Operation column:

- cr specifies consistent reads performed by the row source
- r specifies physical reads performed by the row source
- w specifies physical writes performed by the row source
- time specifies time in microseconds

Wait Event Information

If wait event information exists, the TKPROF output includes a section similar to the following:

```
Elapsed times include waiting on following events:
  Event waited on                             Times   Max. Wait  Total Waited
  ----------------------------------------   Waited  ----------  ------------
  db file sequential read                      8084        0.12          5.34
  direct path write                             834        0.00          0.00
  direct path write temp                        834        0.00          0.05
  db file parallel read                           8        1.53          5.51
  db file scattered read                       4180        0.07          1.45
  direct path read                             7082        0.00          0.05
  direct path read temp                        7082        0.00          0.44
  rdbms ipc reply                                20        0.00          0.01
  SQL*Net message to client                       1        0.00          0.00
  SQL*Net message from client                     1        0.00          0.00
```

In addition, wait events are summed for the entire trace file at the end of the file.

To ensure that wait events information is written to the trace file for the session, run the following SQL statement:

```
ALTER SESSION SET EVENTS '10046 trace name context forever, level 8';
```

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 under the SQL level, such as triggers. For more information, see "Avoiding the Trigger Trap" on page 21-24.

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 21-12, 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 12, "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 21-14 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. A large gap between CPU and elapsed timings indicates Physical I/Os (PIOs).

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.

The columns in Table 21–6 help you identify a row of statistics.

Table 21–6 TKPROF_TABLE Columns for Identifying a Row of Statistics

Column	Description
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 21-12.

```
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 datatypes in the query.

To avoid this problem, perform the conversion yourself.

> **See Also:** "EXPLAIN PLAN Restrictions" on page 12-4 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. 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 example of `TKPROF` output. Portions have been edited out for the sake of brevity.

Sample TKPROF Header

```
TKPROF: Release 10.1.0.0.0 - Beta on Mon Feb 10 14:43:00 2003

(c) Copyright 2001 Oracle Corporation.  All rights reserved.

Trace file: main_ora_27621.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
********************************************************************************
```

Sample TKPROF Body

```
call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.01       0.00          0          0          0           0
Execute      1      0.00       0.00          0          0          0           0
Fetch        0      0.00       0.00          0          0          0           0
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total        2      0.01       0.00          0          0          0           0

Misses in library cache during parse: 1
Optimizer mode: FIRST_ROWS
Parsing user id: 44

Elapsed times include waiting on following events:
  Event waited on                             Times   Max. Wait  Total Waited
  ----------------------------------------   Waited  ----------  ------------
  SQL*Net message to client                       1        0.00          0.00
  SQL*Net message from client                     1       28.59         28.59
********************************************************************************

select condition
from
 cdef$ where rowid=:1

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          2          0           1
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total        3      0.00       0.00          0          2          0           1

Misses in library cache during parse: 1
Optimizer mode: CHOOSE
Parsing user id: SYS   (recursive depth: 1)

Rows     Row Source Operation
-------  ---------------------------------------------------
      1  TABLE ACCESS BY USER ROWID OBJ#(31) (cr=1 r=0 w=0 time=151 us)

********************************************************************************

SELECT last_name, job_id, salary
  FROM employees
WHERE salary =
  (SELECT max(salary) FROM employees)

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.02       0.01          0          0          0           0
Execute      1      0.00       0.00          0          0          0           0
```

```
Fetch        2      0.00       0.00          0         15          0           1
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total        4      0.02       0.01          0         15          0           1

Misses in library cache during parse: 1
Optimizer mode: FIRST_ROWS
Parsing user id: 44

Rows     Row Source Operation
-------  ---------------------------------------------------
      1  TABLE ACCESS FULL EMPLOYEES (cr=15 r=0 w=0 time=1743 us)
      1   SORT AGGREGATE (cr=7 r=0 w=0 time=777 us)
    107    TABLE ACCESS FULL EMPLOYEES (cr=7 r=0 w=0 time=655 us)

Elapsed times include waiting on following events:
  Event waited on                             Times   Max. Wait  Total Waited
  ----------------------------------------   Waited  ----------  ------------
  SQL*Net message to client                       2        0.00          0.00
  SQL*Net message from client                     2        9.62          9.62
********************************************************************************

********************************************************************************
 delete
         from stats$sqltext st
        where (hash_value, text_subset) not in
             (select --+ hash_aj
                     hash_value, text_subset
                from stats$sql_summary ss
               where (   (   snap_id     < :lo_snap
                          or snap_id     > :hi_snap
                         )
                         and dbid            = :dbid
                         and instance_number = :inst_num
                     )
                   or (   dbid            != :dbid
                       or instance_number != :inst_num)
              )

call     count       cpu    elapsed       disk      query    current rows
------- ------  -------- ---------- ---------- ---------- ----------
Parse        1      0.00       0.00          0          0          0          0
Execute      1     29.60      60.68     266984      43776     131172      28144
Fetch        0      0.00       0.00          0          0          0          0
------- ------  -------- ---------- ---------- ---------- ---------- ----------
total        2     29.60      60.68     266984      43776     131172      28144

Misses in library cache during parse: 1
Misses in library cache during execute: 1
Optimizer mode: CHOOSE
Parsing user id: 22

Rows     Row Source Operation
-------  ---------------------------------------------------
      0  DELETE  (cr=43141 r=266947 w=25854 time=60235565 us)
  28144   HASH JOIN ANTI (cr=43057 r=262332 w=25854 time=48830056 us)
  51427    TABLE ACCESS FULL STATS$SQLTEXT (cr=3465 r=3463 w=0 time=865083 us)
 647529    INDEX FAST FULL SCAN STATS$SQL_SUMMARY_PK
                    (cr=39592 r=39325 w=0 time=10522877 us) (object id 7409)

Elapsed times include waiting on following events:
```

```
  Event waited on                             Times   Max. Wait  Total Waited
  ----------------------------------------   Waited  ----------  ------------
  db file sequential read                      8084        0.12          5.34
  direct path write                             834        0.00          0.00
  direct path write temp                        834        0.00          0.05
  db file parallel read                           8        1.53          5.51
  db file scattered read                       4180        0.07          1.45
  direct path read                             7082        0.00          0.05
  direct path read temp                        7082        0.00          0.44
  rdbms ipc reply                                20        0.00          0.01
  SQL*Net message to client                       1        0.00          0.00
  SQL*Net message from client                     1        0.00          0.00
********************************************************************************
```

Sample TKPROF Summary

```
OVERALL TOTALS FOR ALL NON-RECURSIVE STATEMENTS

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        4      0.04       0.01          0          0          0           0
Execute      5      0.00       0.04          0          0          0           0
Fetch        2      0.00       0.00          0         15          0           1
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total       11      0.04       0.06          0         15          0           1

Misses in library cache during parse: 4
Misses in library cache during execute: 1
Elapsed times include waiting on following events:
  Event waited on                             Times   Max. Wait  Total Waited
  ----------------------------------------   Waited  ----------  ------------
  SQL*Net message to client                        6        0.00          0.00
  SQL*Net message from client                      5       77.77        128.88

OVERALL TOTALS FOR ALL RECURSIVE STATEMENTS

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          2          0           1
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total        3      0.00       0.00          0          2          0           1

Misses in library cache during parse: 1
    5  user  SQL statements in session.
    1  internal SQL statements in session.
    6  SQL statements in session.
********************************************************************************
Trace file: main_ora_27621.trc
Trace file compatibility: 9.00.01
Sort options: default
       1  session in tracefile.
       5  user  SQL statements in trace file.
       1  internal SQL statements in trace file.
       6  SQL statements in trace file.
       6  unique SQL statements in trace file.
      76  lines in trace file.
     128  elapsed seconds in trace file.
```

Part V

Real Application Testing

Part V provides information on how to perform real application testing to assure the integrity of database changes.

To help you perform real-world testing of Oracle Database, Oracle offers two Real Application Testing solutions to test the effect of system changes on a production workload without the risks of impacting your production system. Database Replay enables you to replay a full production workload on a test system to assess the overall impact of system changes. The SQL Performance Analyzer enables you to assess the impact of system changes to SQL performance using a SQL Workload.

The chapters in this part include:

- Chapter 22, "Database Replay"
- Chapter 23, "SQL Performance Analyzer"

22

Database Replay

Before system changes are made, such as hardware and software upgrades, extensive testing is usually performed in a test environment to validate the changes. However, despite the testing, the new system often experiences unexpected behavior when it enters production because the testing was not performed using a realistic workload. The inability to simulate a realistic workload during testing is one of the biggest challenges when validating system changes.

Database Replay enables realistic testing of system changes by essentially recreating the production workload environment on a test system. Using Database Replay, you can capture a workload on the production system and replay it on a test system with the exact timing, concurrency, and transaction characteristics of the original workload. This enables you to fully assess the impact of the change, including undesired results, new contention points, or plan regressions. Extensive analysis and reporting is provided to help identify any potential problems, such as new errors encountered and performance divergence.

Capturing the production workload eliminates the need to develop simulation workloads or script, resulting in significant cost reduction and time savings. By using Database Replay, realistic testing of complex applications that previously took months using load simulation tools can now be completed in days. This enables you to rapidly test changes and adopt new technologies with a higher degree of confidence and at lower risk.

Database Replay performs workload capture of external client workload at the database level and has negligible performance overhead. You can use Database Replay to test any significant system changes, including:

- Database and operating system upgrades
- Configuration changes, such as conversion of a database from a single instance to an Oracle RAC environment
- Storage, network, and interconnect changes
- Operating system and hardware migrations

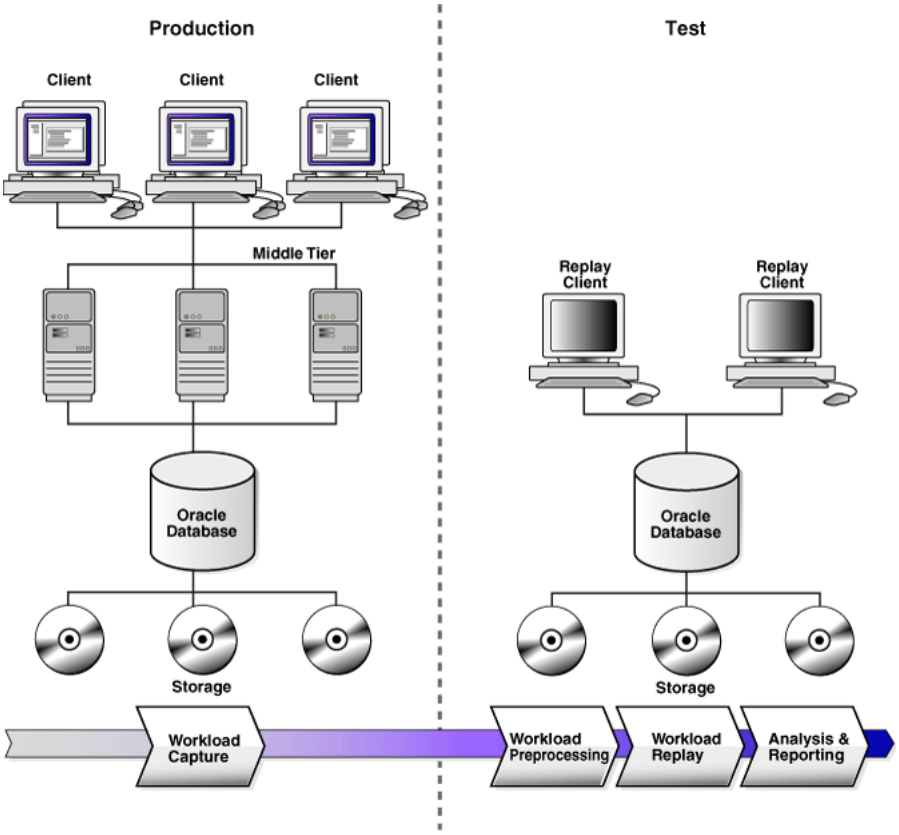
This chapter describes how to use the Database Replay feature of Oracle Database and contains the following sections:

Overview of Database Replay

By capturing a workload on the production system and replaying it on a test system, Database Replay enables you to realistically test system changes by essentially recreating the production workload environment on a test system.

Figure 22–1 illustrates the Database Replay workflow.

Figure 22–1 Database Replay Architecture



Using Database Replay requires four main steps, as shown in Figure 22–1:

- Workload Capture
- Workload Preprocessing
- Workload Replay
- Analysis and Reporting

Workload Capture

The first step in using Database Replay is to capture the production workload. Capturing a workload involves recording all requests made by external clients to Oracle Database. When workload capture is enabled, all external client requests directed to Oracle Database are tracked and stored in binary files, called capture files, on the file system. These capture files are platform independent and can be transported to another system. You can specify the start time and duration for the workload capture, as well as the location to store the capture files. Once workload capture begins, all external database calls are written to the capture files. The capture

files contain all relevant information about the client request, such as SQL text, bind values, and transaction information. Background activities and database scheduler jobs are not captured.

For information about how to capture a workload on the production system, see "Capturing a Database Workload" on page 22-4.

Workload Preprocessing

Once the workload has been captured, the information in the capture files need to be preprocessed. Preprocessing transforms the captured data into replay files and creates all necessary metadata needed for replaying the workload. This must be done once for every captured workload before they can be replayed. After the captured workload is preprocessed, it can be replayed repeatedly on a replay system running the same version of Oracle Database. Typically, the capture files should be copied to another system for preprocessing. As workload preprocessing can be time consuming and resource intensive, it is recommended that this step be performed on the test system where the workload will be replayed.

For information about how to preprocess a captured workload, see "Preprocessing a Database Workload" on page 22-13.

Workload Replay

After a captured workload has been preprocessed, it can be replayed on a test system. During the workload replay phase, Oracle Database performs the actions recorded during the workload capture phase on the test system by recreating all captured external client requests with the same timing, concurrency, and transaction dependencies of the production system.

Database Replay uses a client program called the replay client to recreate all external client requests recorded during workload capture. Depending on the captured workload, you may need one or more replay clients to properly replay the workload. A calibration tool is provided to help determine the number of replay clients needed for a particular workload. Because the entire workload is replayed, including DML and SQL queries, the data in the replay system should be logically similar to the data in the capture system to minimize data divergence and enable reliable analysis of the replay.

For information about how to replay a preprocessed workload on the test system, see "Replaying a Database Workload" on page 22-15.

Analysis and Reporting

Once the workload is replayed, in-depth reporting is provided for you to perform detailed analysis of both workload capture and replay.

The report summary provides basic information about the workload capture and replay, such as errors encountered during replay and data divergence in rows returned by DML or SQL queries. A comparison of several statistics—such as DB time, average active sessions, and user calls—between the workload capture and the workload replay is also provided. For advanced analysis, Automatic Workload Repository (AWR) reports are available to enable detailed comparison of performance statistics between the workload capture and the workload replay. The information available in these reports is very detailed, and some differences between the workload capture and replay can be expected.

For application-level validation, you should consider developing a script to assess the overall success of the replay. For example, if 10,000 orders are processed during workload capture, you should validate that similar number of orders are also processed during replay.

For information about how to analyze data and performance divergence using Database Replay reports, see "Analyzing Replayed Workload" on page 22-30.

Capturing a Database Workload

This section describes how to capture a database workload on the production system. The primary tool for capturing database workloads is Oracle Enterprise Manager. If for some reason Oracle Enterprise Manager is unavailable, you can capture database workloads using APIs.

This section contains the following topics:

Prerequisites for Capturing a Database Workload

Before starting a workload capture, you should have a strategy in place to restore the database on the test system. Before a workload can be replayed, the state of the application data on the replay system should be similar to that of the capture system when replay begins. To accomplish this, consider using one of the following methods:

- Recovery Manager (RMAN) `DUPLICATE` command
- Snapshot standby
- Data Pump Import and Export

This will allow you to restore the database on the replay system to the application state as of the workload capture start time.

See Also:

- *Oracle Database Backup and Recovery User's Guide* for information about duplicating a database using RMAN
- *Oracle Data Guard Concepts and Administration* for information about managing snapshot standby databases
- *Oracle Database Utilities* for information about using Data Pump

Workload Capture Options

Proper planning before workload capture is required to ensure that the capture will be accurate and useful when replayed in another environment.

Before capturing a database workload, carefully consider the following options:

- Restarting the Database
- Defining the Workload Filters
- Setting Up the Capture Directory

Restarting the Database

While this step is not required, Oracle recommends that the database be restarted before capturing the workload to ensure that ongoing and dependent transactions are allowed to be completed or rolled back before the capture begins. If the database is not restarted before the capture begins, transactions that are in progress or have yet to be committed will not be fully captured in the workload. Ongoing transactions will thus not be replayed properly because only the part of the transaction whose calls were captured will be replayed. This may result in undesired data divergence when the workload is replayed. Any subsequent transactions with dependencies on the incomplete transactions may also generate errors during replay.

Before restarting the database, determine an appropriate time to shut down the production database prior to the workload capture time period when it is the least disruptive. For example, you may want to capture a workload that begins at 8:00 a.m. However, to avoid service interruption during normal business hours, you may not want to restart the database at this time. In this case, you should consider starting the workload capture at an earlier time, so that the database can be restarted at a time that is less disruptive.

Once the database is restarted, it is important to start the workload capture before any user sessions reconnect and start issuing any workload. Otherwise, transactions performed by these user sessions will not be replayed properly in subsequent database replays, because only the part of the transaction whose calls were executed after the workload capture is started will be replayed. To avoid this problem, consider restarting the database in `RESTRICTED` mode using `STARTUP_RESTRICTED`, which will only allow the SYS user to login and start the workload capture. By default, once the workload capture begins, any database instance that are in `RESTRICTED` mode will automatically switch to `UNRESTRICTED` mode, and normal operations can continue while the workload is being captured.

> **See Also:** *Oracle Database Administrator's Guide* for information about restricting access to an instance at startup

Only one workload capture can be performed at any given time. For Oracle Real Application Clusters (RAC), workload capture is performed for the entire database. To enable a clean state before starting to capture the workload, all the instances need to be restarted. You can do this by:

1. Shutting down all the instances.
2. Restarting one of the instances.
3. Starting workload capture.
4. Restarting the rest of the instances.

Defining the Workload Filters

By default, all user sessions are recorded during workload capture. You can use workload filters to specify which user sessions to include in or exclude from the workload. Inclusion filters enable you to specify user sessions that will be captured in the workload. This is useful if you want to capture only a subset of the database workload. Exclusion filters enable you to specify user sessions that will not be

captured in the workload. This is useful if you want to filter out session types that do not need to captured in the workload. For example, if the system where the workload will be replayed is running Oracle Enterprise Manager (EM), replaying captured EM sessions on the system will result in duplication of workload. In this case, you may want to use exclusion filters to filter out EM sessions. You can use either inclusion filters or exclusion filters in a workload capture, but not both.

Setting Up the Capture Directory

Determine the location and set up a directory where the captured workload will be stored. Before starting the workload capture, ensure that the directory is empty and has ample disk space to store the workload. If the directory runs out of disk space during a workload capture, the capture will stop.

For Oracle RAC, consider using a shared file system. Alternatively, you can set up capture directory paths that resolve to separate physical directories on each instance, but you will need to collect the capture files created in each of these directories into a single directory before preprocessing the workload capture.

Workload Capture Restrictions

The following types of client requests are not captured in a workload:

- Direct path load of data from external files using utilities such as SQL*Loader
- Shared server requests (Oracle MTS)
- Oracle Streams
- Advanced replication streams
- Non-PL/SQL based Advanced Queuing (AQ)
- Flashback queries
- Oracle Call Interface (OCI) based object navigations
- Non SQL-based object access
- Distributed transactions (any distributed transactions that are captured will be replayed as local transactions)
- Remote `DESCRIBE` and `COMMIT` operations

Capturing a Database Workload Using Enterprise Manager

This section describes how to capture a database workload using Enterprise Manager.

To capture a database workload using Enterprise Manager:

1. On the Software and Support page, under Real Application Testing, click **Database Replay**.

   The Database Replay page appears.

2. In the Go to Task column, click the icon that corresponds to the Capture Workload task.

   The Capture Workload: Plan Environment page appears.

3. Verify that all prerequisites are met before proceeding.

   For information about the prerequisites, see "Prerequisites for Capturing a Database Workload" on page 22-4. For each verified prerequisite, check the box in the Acknowledge column. Once all prerequisites are verified, click **Next**.

The Capture Workload: Options page appears.

4. Select the workload capture options.

   - Under Database Restart Options, select whether the database will be restarted before workload capture.

     It is strongly recommended that the database be restarted before capturing a workload to enable a clean state for workload capture. Otherwise, potential problems may arise when replaying the workload. For more information, see "Restarting the Database" on page 22-5.

   - Under Workload Filters, select whether to use exclusion filters by selecting **Exclusion** in the Filter Mode list, or inclusion filters by selecting **Inclusion** in the Filter Mode list.

     To add filters, click Add Another Row and enter the filter name, session attribute type, and attribute value in the corresponding fields. For more information, see "Defining the Workload Filters" on page 22-5.

   After selecting the desired workload capture options, click **Next**. The Capture Workload: Parameters page appears.

5. Define the parameters for the workload capture.

   - Under Workload Capture Parameters, in the Capture Name field, enter a name for the workload capture. In the Directory Object list, select the directory where the captured workload will be stored. You must select a directory that does not already contain a workload capture. For more information, see "Setting Up the Capture Directory" on page 22-6.

     To create a new directory object, click **Create Directory Object**. The Create Directory Object page appears. In the Name field, enter a name for the directory object. In the Path field, enter the path to the directory object. To test if the directory exists in the file system, click **Test File System**. If the directory does not exist, it will need to be created first.

   - Under Database Shutdown Parameters, select the type of database shutdown method to perform. This option only appears if the database will be restarted before workload capture. The types of available database shutdown methods include:

     – Immediate

       An immediate shutdown will roll back all active transactions and disconnect all connected users prior to shutting down the database.

     – Transactional

       A transactional shutdown will first complete all active transactions and then disconnect the connected user prior to shutting down the database.

     – Abort

       An abort shutdown will shut down the database instantaneously by aborting all active transactions.

   - Under Database Startup Parameters, select if the database will restart using the current default server parameter file (spfile) or a specific parameter file (pfile). To select a pfile, enter the fully qualified name for the pfile. This option only appears if the database will be restarted before workload capture.

   After defining the parameters for the workload capture, click **Next**. The Capture Workload: Schedule page appears.

6. Under Job Parameters, define the parameters for the job:
   - In the Job Name field, enter a name for the job name or accept the system generated name.
   - In the Description field, enter an optional description of the job.
7. Under Job Schedule, specify a start time and duration for the workload capture:
   - Under Start, select whether the job will run immediately by selecting **Immediately**, or at a later time by selecting **Later** and specifying the desired time using the Date and Time fields.
   - Under Capture Duration, specify how long the job will run by selecting **Duration** and specifying the desired duration using the Hours and Minutes fields. To not specify a capture duration, select **Not Specified**. If a capture duration is unspecified, the job must be stopped manually.
8. Under Job Credentials, enter the host and database login credentials:
   - Under Host Credentials, enter the username and password for the host machine.
   - Under Database Credentials, enter the username and password for the database that will used for the workload capture. The user needs the `DBA` privilege in order to capture the workload.

   Click **Next**. The Capture Workload: Review page appears.
9. Review the job settings for the workload capture that have been defined. To run the job, click **Submit**. To make changes, click **Back**. To cancel the workload capture without saving changes, click **Cancel**.
10. Depending on the job settings that have been defined:
    - If the job is scheduled to start immediately and the database will be restarted, the Confirmation: Restart Database page appears. To restart the database, click **Yes**. The Information: Restart Database page appears while the database is being restarted. Once the database is restarted, the workload capture begins automatically. Click **Refresh**. The View Workload Capture page appears.
    - If the job is scheduled to start immediately but the database will not be restarted, the workload capture begins automatically and the View Workload Capture page appears.
    - If the job is scheduled to start at a later time, the Database Replay page appears with a confirmation that the job has been successfully created.

    Once workload capture begins, you can monitor the capture process using the View Workload Capture page, as described in "Monitoring an Active Workload Capture" on page 22-9.

    Tip: After capturing a workload on the production system, you need to preprocess the captured workload, as described in "Preprocessing a Database Workload" on page 22-13.

Monitoring Workload Capture Using Enterprise Manager

This section describes how to monitor workload capture using Enterprise Manager. The primary tool for monitoring workload capture is Oracle Enterprise Manager. Using Enterprise Manager, you can:

- Monitor or stop an active workload capture

- View or delete a completed workload capture

If for some reason Oracle Enterprise Manager is unavailable, you can monitor workload capture using views, as described in "Monitoring Workload Capture Using Views" on page 22-13.

This section contains the following topics:

Monitoring an Active Workload Capture

This section describes how to monitor an active workload capture using Enterprise Manager.

To monitor an active workload capture:

1. On the Software and Support page, under Real Application Testing, click **Database Replay**.

   The Database Replay page appears.

2. Under Active Capture and Replay, select the workload capture you want to monitor and click **View**.

   The View Workload Capture page appears.

3. Under Summary, information about the workload capture is displayed.

4. To view the workload profile, click the **Workload Profile** tab.

   Under Average Active Sessions, the Active Sessions chart provides a graphic display of the captured session activity compared to the uncaptured session activity (such as background activities or filtered sessions).

   Under Comparison, various statistics for the workload capture are displayed, including database time, average active sessions, user calls, transactions, connects, and application errors. The statistics for the total session activity are displayed in the Total column, and the statistics for the captured session activity are displayed in the Capture column. The Percentage of Total column displays the percentage of total session activity that are being captured in the workload.

   To view the workload capture report, click **View Workload Capture Report**.

5. To view workload filters used by the workload capture, click the **Workload Filters** tab.

   Details about the workload filters used by the workload capture are displayed, including the workload filter name, type, session attribute, and value.

6. To return to the Database Replay page, click **OK**.

Stopping an Active Workload Capture

This section describes how to stop an active workload capture using Enterprise Manager.

To stop an active workload capture:

1. On the Software and Support page, under Real Application Testing, click **Database Replay**.

   The Database Replay page appears.

2. Under Active Capture and Replay, select the workload capture you want to stop and click **Stop**.

   The Confirmation page appears.

3. To confirm that you want to stop the workload capture, click **Yes**.

   The Export AWR Data page appears.

4. To export the Automatic Workload Repository (AWR) data, click **Yes**.

   Exporting AWR data enables detailed analysis of the workload. This data is also required if you plan to run the AWR Compare Period report on a pair of workload captures or replays. If you choose not to export AWR data, click **No**. You can still export AWR data from a completed workload capture at a later time from the View Workload Capture History page. For information about the AWR, see "Overview of the Automatic Workload Repository" on page 5-8.

Managing a Completed Workload Capture

This section describes how to manage a completed workload capture using Enterprise Manager.

To manage a completed workload capture:

1. On the Software and Support page, under Real Application Testing, click **Database Replay**.

   The Database Replay page appears.

2. Click **View Workload Capture History**.

   The View Workload Capture History page appears.

3. To delete a workload capture, select the workload capture and click **Delete**.

4. To export AWR data for a workload capture, select the workload capture and click **Export AWR Data**.

   Exporting AWR data enables detailed analysis of the workload. This data is also required if you plan to run the AWR Compare Period report on a pair of workload captures or replays.

5. To view details about a workload capture, select the workload capture and click **View**.

   The View Workload Capture page appears.

6. Under Summary, information about the workload capture is displayed.

7. To view the workload profile, click the **Workload Profile** tab.

   Under Average Active Sessions, the Active Sessions chart provides a graphic display of the captured session activity compared to the uncaptured session activity (such as background activities or filtered sessions). This chart will be shown only when there is Active Session History (ASH) data available for the capture period. For information about ASH, see "Active Session History" on page 5-3.

   Under Comparison, various statistics for the workload capture are displayed, including database time, average active sessions, user calls, transactions, connects, and application errors. The statistics for the total session activity are displayed in the Total column, and the statistics for the captured session activity are displayed in the Capture column. The Percentage of Total column displays the percentage of total session activity that are being captured in the workload.

To view the workload capture report, click **View Workload Capture Report**.

8. To view workload filters used by the workload capture, click the **Workload Filters** tab.

   Details about the workload filters used by the workload capture are displayed, including the workload filter name, type, session attribute, and value.

9. To return to the Database Replay page, click **OK**.

Capturing a Database Workload Using APIs

This section describes how to capture a database workload using APIs. Capturing a database workload using the `DBMS_WORKLOAD_CAPTURE` package involves:

- Adding and Removing Workload Filters
- Starting a Workload Capture
- Stopping a Workload Capture
- Exporting AWR Data for Workload Capture

See Also: *Oracle Database PL/SQL Packages and Types Reference*

Adding and Removing Workload Filters

This section describes how to add and remove workload filters. For information about using workload filters, see "Defining the Workload Filters" on page 22-5.

To add filters to a workload capture, use the `ADD_FILTER` procedure:

```
BEGIN
  DBMS_WORKLOAD_CAPTURE.ADD_FILTER (
                           fname => 'user_ichan',
                           fattribute => 'USER',
                           fvalue => 'ICHAN');
END;
/
```

In this example, the `ADD_FILTER` procedure adds a filter named user_ichan, which can be used to filter out all sessions belonging to the user name ICHAN.

The `ADD_FILTER` procedure in this example uses the following parameters:

- The `fname` required parameter specifies the name of the filter that will be added.
- The `fattribute` required parameter specifies the attribute on which the filter will be applied. Valid values include PROGRAM, MODULE, ACTION, SERVICE, INSTANCE_NUMBER, and USER.
- The `fvalue` required parameter specifies the value for the corresponding attribute on which the filter will be applied. It is possible to use wildcards such as % with some of the attributes, such as modules and actions.

To remove filters from a workload capture, use the `DELETE_FILTER` procedure:

```
BEGIN
  DBMS_WORKLOAD_CAPTURE.DELETE_FILTER (fname => 'user_ichan');
END;
/
```

In this example, the `DELETE_FILTER` procedure removes the filter named user_ichan from the workload capture.

The DELETE_FILTER procedure in this example uses the fname required parameter, which specifies the name of the filter to be removed.

Starting a Workload Capture

Before starting a workload capture, you must first complete the prerequisites for capturing a database workload, as described in "Prerequisites for Capturing a Database Workload" on page 22-4. You should also review the workload capture options, as described in "Workload Capture Options" on page 22-4.

It is important to have a well-defined starting point for the workload so that the replay system can be restored to that point before initiating a replay of the captured workload. To have a well-defined starting point for the workload capture, it is preferable not to have any active user sessions when starting a workload capture. If active sessions perform ongoing transactions, those transactions will not be replayed properly in subsequent database replays, since only that part of the transaction whose calls were executed after the workload capture is started will be replayed. To avoid this problem, consider restarting the database in RESTRICTED mode using STARTUP_RESTRICTED prior to starting the workload capture. Once the workload capture begins, the database will automatically switch to UNRESTRICTED mode and normal operations can continue while the workload is being captured. For more information about restarting the database before capturing a workload, see "Restarting the Database" on page 22-5.

To start the workload capture, use the START_CAPTURE procedure:

```
BEGIN
  DBMS_WORKLOAD_CAPTURE.START_CAPTURE (name => 'dec06_peak',
                           dir => 'dec06',
                           duration => 600);
END;
/
```

In this example, a workload named dec06_peak will be captured for 600 seconds and stored in the operating system defined by the database directory object named dec06.

The START_CAPTURE procedure in this example uses the following parameters:

- The name required parameter specifies the name of the workload that will be captured.
- The dir required parameter specifies a directory object pointing to the directory where the captured workload will be stored.
- The duration optional parameter specifies the number of seconds before the workload capture will end. If a value is not specified, the workload capture will continue until the FINISH_CAPTURE procedure is called.

Stopping a Workload Capture

To stop the workload capture, use the FINISH_CAPTURE procedure:

```
BEGIN
  DBMS_WORKLOAD_CAPTURE.FINISH_CAPTURE ();
END;
/
```

In this example, the FINISH_CAPTURE procedure finalizes the workload capture and returns the database to a normal state.

Tip: After capturing a workload on the production system, you need to preprocess the captured workload, as described in "Preprocessing a Database Workload" on page 22-13.

Exporting AWR Data for Workload Capture

Exporting AWR data enables detailed analysis of the workload. This data is also required if you plan to run the AWR Compare Period report on a pair of workload captures or replays.

To export AWR data, use the `EXPORT_AWR` procedure:

```
BEGIN
  DBMS_WORKLOAD_CAPTURE.EXPORT_AWR (capture_id => 2);
END;
/
```

In this example, the AWR snapshots that correspond to the workload capture with a capture ID of 2 are exported. The `EXPORT_AWR` procedure uses the `capture_id` required parameter, which specifies the ID of the capture whose AWR snapshots will be exported. This procedure will work only if the corresponding workload capture was performed in the current database and the AWR snapshots that correspond to the original capture time period are still available.

Monitoring Workload Capture Using Views

This section summarizes the views that you can display to monitor workload capture. You need DBA privileges to access these views.

- The `DBA_WORKLOAD_CAPTURES` view lists all the workload captures that have been captured in the current database.
- The `DBA_WORKLOAD_FILTERS` view lists all workload filters used for workload captures defined in the current database.

See Also: *Oracle Database Reference* for information on these views

Preprocessing a Database Workload

After a workload is captured and setup of the test system is complete, the captured data must be preprocessed. Preprocessing a captured workload transforms the captured data into replay files and creates all necessary metadata. This must be done once for every captured workload before they can be replayed. After the captured workload is preprocessed, it can be replayed repeatedly on a replay system.

To preprocess a captured workload, you will first need to move all captured data files from the directory where they are stored on the capture system to a directory on the instance where the preprocessing will be performed. Preprocessing is resource intensive and should be performed on a system that is:

- Separate from the production system
- Running the same version of Oracle Database as the replay system

For Oracle RAC, select one database instance of the replay system for the preprocessing. This instance must have access to the captured data files that require preprocessing, which can be stored on a local or shared file system. If the capture directory path on the capture system resolves to separate physical directories in each instance, you will need to move all the capture files created in each of these directories into a single directory on which preprocessing will be performed.

Typically, you will preprocess the captured workload on the replay system. If you plan to preprocess the captured workload on a system that is separate from the replay system, you will also need to move all preprocessed data files from the directory where they are stored on the preprocessing system to a directory on the replay system after preprocessing is complete.

The primary tool for preprocessing workload captures is Oracle Enterprise Manager. If for some reason Oracle Enterprise Manager is unavailable, you can preprocess workload captures using the APIs.

This section contains the following topics:

- Preprocessing a Captured Workload Using Enterprise Manager
- Preprocessing a Captured Workload Using APIs

Tip: Before you can preprocess a captured workload, you need to:

- Capture the workload on the production system, as described in "Capturing a Database Workload" on page 22-4

Preprocessing a Captured Workload Using Enterprise Manager

This section describes how to preprocess a captured workload using Enterprise Manager.

To preprocess a captured workload using Enterprise Manager:

1. On the Software and Support page, under Real Application Testing, click **Database Replay**.

   The Database Replay page appears.

2. In the Go to Task column, click the icon that corresponds to the Preprocess Captured Workload task.

   The Preprocess Captured Workload page appears.

3. In the Directory Object list, select a directory that contains the captured workload that you want to preprocess.

   After a directory is selected, the Preprocess Captured Workload page will be refreshed to display the Capture Summary section, which contains information about the captured workload in the selected directory.

   To view additional details about the captured workload, expand **Capture Details**. The expanded Capture Details section displays the workload profile and details for the captured workload.

4. Click **Preprocess Workload**.

   The Preprocess Captured Workload: Database Version page appears.

5. Ensure that the current database version displayed matches the database version on the intended replay system and click **Next**.

   Preprocessing must be performed on a system that is running the same version of Oracle Database as the replay system.

   The Preprocess Captured Workload: Schedule page appears.

6. Define the parameters for the preprocessing job.

   - Under Job Parameters, enter a name and a description for the job.

- Under Start, select whether the job will run immediately by selecting **Immediately**, or at a later time by selecting **Later** and specifying the desired time using the Date and Time fields.
- Under Host Credentials, enter the user name and password information for the database host that will used for the preprocessing.

After defining the job parameters, click **Next**. The Preprocess Captured Workload: Review page appears.

7. Review the selected options for the preprocessing job. To preprocess the captured workload, click **Submit**. To make changes, click **Back**. To cancel preprocessing without saving changes, click **Cancel**.

> **Tip:** After preprocessing a captured workload, you can replay it on the test system, as described in "Replaying a Database Workload" on page 22-15.

Preprocessing a Captured Workload Using APIs

This section describes how to preprocess a captured workload using the `DBMS_WORKLOAD_REPLAY` package.

> **See Also:** *Oracle Database PL/SQL Packages and Types Reference*

To preprocess a captured workload, use the `PROCESS_CAPTURE` procedure:

```
BEGIN
  DBMS_WORKLOAD_REPLAY.PROCESS_CAPTURE (capture_dir => 'dec06');
END;
/
```

In this example, the captured workload stored in the dec06 directory will be preprocessed.

The `PROCESS_CAPTURE` procedure in this example uses the `capture_dir` required parameter, which specifies the directory that contains the captured workload to be preprocessed.

> **Tip:** After preprocessing a captured workload, you can replay it on the test system, as described in "Replaying a Database Workload" on page 22-15.

Replaying a Database Workload

This section describes how to replay a database workload on the test system. After a captured workload is preprocessed, it can be replayed repeatedly on a replay system that is running the same version of Oracle Database. The primary tool for replaying database workloads is Oracle Enterprise Manager. However, you can also replay database workloads using APIs.

This section contains the following topics:

- Monitoring Workload Replay Using Views

Tip: Before you can replay a workload, you must first:

- Capture the workload on the production system, as described in "Capturing a Database Workload" on page 22-4
- Preprocess the captured workload, as described in "Preprocessing a Database Workload" on page 22-13

Setting Up the Test System

Typically, the replay system where the preprocessed workload will be replayed should be a test system that is separate from the production system. Before a test system can be used for replay, it must be prepared properly as described in the following sections:

- Restoring the Database
- Resetting the System Time

Restoring the Database

Before a workload can be replayed, the application data state should be logically equivalent to that of the capture system at the start time of workload capture. This minimizes data divergence during replay. The method for restoring the database depends on the backup method that was used before capturing the workload. For example, if RMAN was used to back up the capture system, you can use RMAN `DUPLICATE` capabilities to create the test database. For more information, see "Prerequisites for Capturing a Database Workload" on page 22-4.

After the database is created with the appropriate application data on the test system, perform the system change you want to test, such as a database or operating system upgrade. The primary purpose of Database Replay is to test the effect of system changes on a captured workload. Therefore, the system changes you make should define the test you are conducting with the captured workload.

Resetting the System Time

It is recommended that the system time on the replay system host be changed to a value that approximately matches the capture start time just before replay is started. Otherwise, an invalid data set may result when replaying time-sensitive workloads. For example, a captured workload that contains SQL statements using the `SYSDATE` and `SYSTIMESTAMP` functions may cause data divergence when replayed on a system that has a different system time. Resetting the system time will also minimize job scheduling inconsistencies between capture and replay.

Steps for Replaying a Database Workload

Proper planning of the workload replay ensures that the replay will be accurate. The section describes the required steps for replaying a database workload:

- Setting Up the Replay Directory
- Resolving References to External Systems
- Remapping Connections
- Specifying Replay Options
- Setting Up Replay Clients

Setting Up the Replay Directory

The captured workload must have been preprocessed and copied to the replay system. A directory object for the directory to which the preprocessed workload is copied must exist in the replay system.

Resolving References to External Systems

A captured workload may contain references to external systems, such as database links or external tables. Typically, you should disable or reconfigure these external interactions to avoid impacting other production systems during replay. External references that need to be resolved before replay a workload include:

- Database links

  It is typically not desirable for the replay system to interact with other databases. Therefore, you should disable or reconfigure all database links to point to an appropriate database that contains the data needed for replay.

- External tables

  All external files specified using directory objects referenced by external tables need to be available to the database during replay. The content of these files should be the same as during capture, and the filenames and directory objects used to define the external tables should also be valid.

- Directory objects

  You should disable or reconfigure any references to directories on the production system by appropriately redefining the directory objects present in the replay system after restoring the database.

- URLs

  URLs need to be configured so that Web services accessed during the workload capture will point to the proper URLs during replay.

- E-mails

  To avoid resending E-mail notifications during replay, the mail server on the replay system should be configured to ignore requests for outgoing E-mails.

To avoid impacting other production systems during replay, Oracle strongly recommends running the replay within an isolated private network that does not have access to the production environment hosts.

Remapping Connections

During workload capture, connection strings used to connect to the production system are captured. In order for the replay to succeed, you need to remap these connection strings to the replay system. The replay clients can then connect to the replay system using the remapped connections.

For Oracle RAC databases, you can map all connection strings to a load balancing connection string. This is especially useful if the number of nodes on the replay system is different from the capture system. Alternatively, if you wish to direct workload to specific instances, you can use services or explicitly specify the instance identifier in the remapped connection strings.

Specifying Replay Options

After the database is restored and connections are remapped, you can set the following replay options as appropriate:

- Preserving COMMIT Order
- Controlling Session Logins
- Controlling Think Time

Preserving COMMIT Order The `synchronization` parameter controls whether the `COMMIT` order in the captured workload will be preserved during replay. By default, this option is enabled to preserve the `COMMIT` order in the captured workload during replay. All transactions will be executed only after all dependent transactions have been committed.

You can disable this option, but the replay will likely yield significant data divergence. However, this may be desirable if the workload consists primarily of independent transactions, and divergence during unsynchronized replay is acceptable

Controlling Session Logins The `connect_time_scale` parameter enables you to scale the elapsed time between the time when the workload capture began and when sessions connect. You can use this option to manipulate the session connect time during replay with a given percentage value. The default value is 100, which will attempt to connect all sessions as captured. Setting this parameter to 0 will attempt to connect all sessions immediately.

Controlling Think Time User think time is the elapsed time while the user waits between issuing calls. To control replay speed, use the `think_time_scale` parameter to scale user think time during replay. If user calls are being executed slower during replay than during capture, you can make the database replay attempt to catch up by setting the `think_time_auto_correct` parameter to `TRUE`. This will make the replay client shorten the think time between calls, so that the overall elapsed time of the replay will more closely match the captured elapsed time.

Setting Up Replay Clients

The replay client is a multithreaded program (an executable named wrc located in the `$ORACLE_HOME/bin` directory) where each thread submits a workload from a captured session. Before replay begins, the database will wait for replay clients to connect. At this point, you need to set up and start the replay clients, which will connect to the replay system and send requests based on what has been captured in the workload.

Before starting replay clients, ensure that the:

- Replay client software is installed on the hosts where it will run
- Replay clients have access to the replay directory
- Replay directory contains the preprocessed workload capture
- Replay user has the correct user ID, password, and privileges (the replay user needs the DBA role and cannot be the `SYS` user)

After these prerequisites are met, you can proceed to set up and start the replay clients using the wrc executable. The wrc executable uses the following syntax:

```
wrc [user/password[@server]] MODE=[value] [keyword=[value]]
```

The parameters `user` and `password` specify the username and password used to connect to the host where the wrc executable is installed. The parameter `server` specifies the server where the wrc executable is installed. The parameter `mode` specifies the mode in which to run the wrc executable. Possible values include `replay` (the default), `calibrate`, and `list_hosts`. The parameter `keyword` specifies the

options to use for the execution and is dependent on the mode selected. To display the possible keywords and their corresponding values, run the wrc executable without any arguments.

The following sections describe the modes that you can select when running the wrc executable:

- Calibrating Replay Clients
- Starting Replay Clients
- Displaying Host Information

Calibrating Replay Clients Since one replay client can initiate multiple sessions with the database, it is not necessary to start a replay client for each session that was captured. The number of replay clients that need to be started depends on the number of workload streams, the number of hosts, and the number of replay clients for each host.

To estimate the number of replay clients and hosts that are required to replay a particular workload, run the wrc executable in calibrate mode.

In calibration mode, the wrc executable accepts the following keywords:

- `replaydir` specifies the directory that contains the preprocessed workload capture you want to replay. If unspecified, it defaults to the current directory.
- `process_per_cpu` specifies the maximum number of client processes that can run per CPU. The default value is 4.
- `threads_per_process` specifies the maximum number of thread that can run within a client process. The default value is 50.

The following example shows how to run the wrc executable in calibrate mode:

```
%> wrc mode=calibrate replaydir=./replay
```

In this example, the wrc executable is executed to estimate the number of replay clients and hosts that are required to replay the workload capture stored in a subdirectory named `replay` under the current directory. In the following sample output, the recommendation is to use at least 21 replay clients divided among 6 CPUs:

```
Workload Replay Client: Release 11.1.0.6.0 - Production on Thu Jul 12
14:06:33 2007

Copyright (c) 1982, 2007, Oracle.  All rights reserved.

Report for Workload in: /oracle/replay/
-----------------------

Recommendation:
Consider using at least 21 clients divided among 6 CPU(s).

Workload Characteristics:
- max concurrency: 1004 sessions
- total number of sessions: 1013

Assumptions:
- 1 client process per 50 concurrent sessions
- 4 client process per CPU
- think time scale = 100
- connect time scale = 100
- synchronization = TRUE
```

Starting Replay Clients After determining the number of replay clients that are needed to replay the workload, you need to start the replay clients by running the wrc executable in replay mode on the hosts where they are installed. Once started, each replay client will initiate one or more sessions with the database to drive the workload replay.

In replay mode, the wrc executable accepts the following keywords:

- `userid` and `password` specify the user ID and password of a replay user for the replay client. If unspecified, these values default to the `system` user.
- `server` specifies the connection string that is used to connect to the replay system. If unspecified, the value defaults to an empty string.
- `replaydir` specifies the directory that contains the preprocessed workload capture you want to replay. If unspecified, it defaults to the current directory.
- `workdir` specifies the directory where the client logs will be written. This parameter is only used in conjunction with the `debug` parameter for debugging purposes.
- `debug` specifies whether debug data will be created. Possible values include:
  - `files`

    Debug data will be written to files in the working directory
  - `stdout`

    Debug data will be written to `stdout`
  - `both`

    Debug data will be written to both files in the working directory and to stdout
  - `none`

    No debug data will be written (the default value)
- `connection_override` specifies whether to override the connection mappings stored in the `DBA_WORKLOAD_CONNECTION_MAP` view. If set to `TRUE`, connection remappings stored in the `DBA_WORKLOAD_CONNECTION_MAP` view will be ignored and the connection string specified using the `server` parameter will be used. If set to `FALSE`, all replay threads will connect using the connection remappings stored in the `DBA_WORKLOAD_CONNECTION_MAP` view. This is the default setting.

The following example shows how to run the wrc executable in replay mode:

```
%> wrc system/oracle@test mode=replay replaydir=./replay
```

In this example, the wrc executable starts the replay client to replay the workload capture stored in a subdirectory named `replay` under the current directory.

After all replay clients have connected, the database will automatically distribute workload capture streams among all available replay clients. At this point, workload replay can begin. After the replay finishes, all replay clients will disconnect automatically.

Displaying Host Information You can display the hosts that participated in a workload capture and workload replay by running the wrc executable in list_hosts mode.

In list_hosts mode, the wrc executable accepts the keyword `replaydir`, which specifies the directory that contains the preprocessed workload capture you want to replay. If unspecified, it defaults to the current directory.

The following example shows how to run the wrc executable in list_hosts mode:

```
%> wrc mode=list_hosts replaydir=./replay
```

In this example, the wrc executable is executed to list all hosts that participated in capturing or replaying the workload capture stored in a subdirectory named `replay` under the current directory. In the following sample output, the hosts that participated in the workload capture and three subsequent replays are shown:

```
Workload Replay Client: Release 11.1.0.6.0 - Production on Thu Jul 12 13:44:48
2007

Copyright (c) 1982, 2007, Oracle.  All rights reserved.

Hosts found:
Capture:
        prod1
        prod2
Replay 1:
        test1
Replay 2:
        test1
        test2
Replay 3:
        testwin
```

Replaying a Database Workload Using Enterprise Manager

This section describes how to replay a database workload using Enterprise Manager.

To replay a database workload using Enterprise Manager:

1. On the Software and Support page, under Real Application Testing, click **Database Replay**.

   The Database Replay page appears.

2. In the Go to Task column, click the icon that corresponds to the Replay Workload task.

   The Replay Workload page appears.

3. In the Directory Object list, select a directory that contains the preprocessed workload that you want to replay.

   After a directory is selected, the Replay Workload page will be refreshed to display the Capture Summary and the Replay History sections. For more information, see "Setting Up the Replay Directory" on page 22-17.

   The Capture Summary section displays information about the preprocessed workload capture in the selected directory. To view additional details about the workload capture, expand **Capture Details**. The expanded Capture Details section displays the workload profile and workload filters used during the workload capture.

4. Click **Set Up Replay**.

   The Replay Workload: Prerequisites page appears.

5. Verify that all prerequisites are met before proceeding.

If you are replaying the workload on a test system, ensure that the test system is properly prepared for replay. For more information, see "Setting Up the Test System" on page 22-16.

Once all prerequisites are completed, click **Continue**.

The Replay Workload: References to External Systems page appears.

6. Verify potential references to all external systems and modify any invalid references.

   Use the links available on the Replay Workload: References to External Systems page to verify the database links, directory objects, and Oracle Streams that may be referenced during the workload capture process. There may be other references to external systems that are not included in these categories. For more information, see "Resolving References to External Systems" on page 22-17.

   Once all references to external systems have been verified and modified as necessary, click **Continue**.

   The Replay Workload: Choose Initial Options page appears.

7. In the Replay Name field, you may enter a name for the replay, or simply use the name generated by the system.

8. Under Initial Options, select whether to use default replay options or replay options from a previous replay (if one is available). If more than one previous replay exist, select the replay you want to use from the Replay Name list.

   Click **Next**. The Replay Workload: Customize Options page appears.

9. Remap captured connection strings to connection strings that point to the replay system.

   Click the **Connection Mappings** tab. There are several methods you can use to remap captured connection strings. You can choose to:

   - **Use a single connect descriptor for all client connections** by selecting this option and entering the connect descriptor you want to use. The connect descriptor should point to the replay system.

     To test the connection, click **Test Connection**. If the connect descriptor is valid, an Information message is displayed to inform you that the connection was successful.

   - **Use a single TNS net service name for all client connections** by selecting this option and entering the net service name you want to use. All replay clients must be able to resolve the net service name, which can be done using a local tnsnames.ora file.

   - **Use a separate connect descriptor or net service name for each client connect descriptor captured in the workload** by selecting this option and, for each capture system value, entering a corresponding replay system value that will be used by the replay client.

   For more information, see "Remapping Connections" on page 22-17.

10. Specify the replay options using the replay parameters.

    To modify the replay behavior, click the **Replay Parameters** tab and enter the desired values for each replay parameter. Using the default values is recommended. For information about setting the replay parameters, see "Specifying Replay Options" on page 22-17.

After setting the replay parameters, click **Next**. The Replay Workload: Prepare Replay Clients page appears.

11. Ensure that replay clients are prepared for replay.

    Before proceeding, the replay clients need to be prepared. For more information, see "Setting Up Replay Clients" on page 22-18.

    After all replay clients are ready to start, click **Next**. The Replay Workload: Wait for Client Connections page appears.

12. Start the replay clients.

    For information about starting replay clients, see "Setting Up Replay Clients" on page 22-18. As replay clients are started, the replay client connections will be displayed under Client Connections. When all replay clients have connected, click **Next**.

    The Replay Workload: Review page appears.

13. Review the options and parameters that have been defined for the workload replay.

    To begin replay, click **Submit**. If no replay clients are connected, this button will be disabled. To make changes, click **Back**. To cancel replay without saving changes, click **Cancel**.

    Once the replay is started, the View Workload Replay page appears. For information about monitoring an active workload replay, see "Monitoring an Active Workload Replay" on page 22-23.

Monitoring Workload Replay Using Enterprise Manager

This section describes how to monitor workload replay using Enterprise Manager. The primary tool for monitoring workload replay is Oracle Enterprise Manager. Using Enterprise Manager, you can:

- Monitor or stop an active workload replay
- View a completed workload replay

If for some reason Oracle Enterprise Manager is unavailable, you can monitor workload replay using views, as described in "Monitoring Workload Replay Using Views" on page 22-29.

This section contains the following topics:

Monitoring an Active Workload Replay

This section describes how to monitor an active workload replay using Enterprise Manager.

To monitor an active workload replay:

1. On the Software and Support page, under Real Application Testing, click **Database Replay**.

   The Database Replay page appears.

2. Under Active Capture and Replay, select the workload replay you want to monitor and click **View**.

The View Workload Replay page appears.

3. Under Summary, information about the workload replay is displayed.
4. To view the workload profile, click the **Workload Profile** tab.

   Under Elapsed Time Comparison, the chart shows how much time it has taken to replay the same workload compared to the elapsed time during the workload capture. If the Replay bar is shorter than the Capture bar, the replay system is processing the workload faster than the capture system.

   Under Divergence, any error and data discrepancies between the replay system and the capture system are displayed. This information can be used as a measure of the replay quality.

   To view a detailed comparison of the workload during capture and replay, expand **Detailed Comparison**. This section displays the following information:

   - Duration

     The duration that was captured in a workload is compared to the amount of time it is taking to replay the workload. In the Capture column, the duration of the time period that was captured is shown. In the Replay column, the amount of time it is taking to replay the workload is shown. The Percentage of Capture column shows the percentage of the captured duration that it is taking to replay the workload. If the value is under 100 percent, the replay system is processing the workload faster than the capture system. If the value is over 100 percent, the replay system is processing the workload slower than the capture system.

   - Database time

     The database time that is consumed in the time period that was captured is compared to the amount of database time that is being consumed while replaying the workload.

   - Average active sessions

     The number of average active sessions captured in the workload is compared to the number of average active session that are being replayed.

   - User calls

     The number of user calls captured in the workload is compared to the number of user calls that are being replayed.

   To view the workload capture report, click **View Workload Replay Report**.
5. To view the connection strings used in the capture and the replay systems, click the **Connection Mappings** tab.
6. To view replay parameters used by the workload replay, click the **Replay Parameters** tab.
7. To stop the workload replay, click **Stop Replay**.
8. To return to the Database Replay page, click **OK**.

Viewing a Completed Workload Replay

This section describes how to view a completed workload replay using Enterprise Manager.

To view a completed workload replay:

1. On the Software and Support page, under Real Application Testing, click **Database Replay**.

   The Database Replay page appears.

2. In the Go to Task column, click the icon that corresponds to the Replay Workload task.

   The Replay Workload page appears.

3. In the Directory Object list, select a directory that contains the preprocessed workload that was used for the replay.

   After a directory is selected, the Replay Workload page will be refreshed to display the Capture Summary and the Replay History sections.

4. The Replay History section displays previous replays of the workload capture. To view details about a previous replay, select the replay and click **View**.

   The View Workload Replay page appears.

5. Under Summary, information about the workload replay is displayed.

6. To view the workload profile, click the **Workload Profile** tab.

   Under Elapsed Time Comparison, the chart shows how much time it has taken to replay the same workload compared to the elapsed time during the workload capture. If the Replay bar is shorter than the Capture bar, the replay system is processing the workload faster than the capture system.

   Under Divergence, any error and data discrepancies between the replay system and the capture system are displayed. This information can be used as a measure of the replay quality.

   To view a detailed comparison of the workload during capture and replay, expand **Detailed Comparison**. This section displays the following information:

   - Duration

     The duration that was captured in a workload is compared to the amount of time it took to replay the workload. In the Capture column, the duration of the time period that was captured is shown. In the Replay column, the amount of time it took to replay the workload is shown. The Percentage of Capture column shows the percentage of the captured duration that it took to replay the workload. If the value is under 100 percent, the replay system processed the workload faster than the capture system. If the value is over 100 percent, the replay system processed the workload slower than the capture system.

   - Database time

     The database time that is consumed in the time period that was captured is compared to the amount of database time that is consumed when replaying the workload.

   - Average active sessions

     The number of average active sessions captured in the workload is compared to the number of average active session that are replayed.

   - User calls

     The number of user calls captured in the workload is compared to the number of user calls that are replayed.

   To view the workload capture report, click **View Workload Replay Report**.

7. To view the connection strings used in the capture and the replay systems, click the **Connection Mappings** tab.
8. To view replay parameters used by the workload replay, click the **Replay Parameters** tab.
9. To run a report, click the **Report** tab.

   There are several types of reports you can run for a completed workload replay:

   - Workload Replay

     The Workload Replay report contains information that can be used to measure data and performance divergence between the capture system and the replay system. To run this report, under Workload Replay Report, click **Run Report**. For information about using the Workload Replay report, see "Using a Workload Replay Report" on page 22-33.

   - AWR Compare Period

     The AWR Compare Period report can be used to compare the AWR data in one workload capture or replay with another. Before running this report, AWR data for the captured or replayed workload must have been previously exported. To run this report, under AWR Compare Period Report, select the first and second workload captures or replays you want to compare and click **Run Report**. If AWR data is not previously exported from the captured or replayed workload, you will be prompted to import the AWR data before continuing. For more information about the AWR Compare Period report, see "Generating Automatic Workload Repository Compare Periods Reports" on page 5-25.

   - AWR

     The AWR report shows the AWR data contained in a workload that was captured or replayed. Before running this report, AWR data must have been previously exported from the captured or replayed workload. To run this report, under AWR Report, select the workload capture or replay for which you want to generate an AWR report and click **Run Report**. If AWR data is not previously exported from the captured or replayed workload, you will be prompted to import the AWR data before continuing. For more information about the AWR report, see "Generating Automatic Workload Repository Reports" on page 5-21.

   - ASH

     The ASH report contains active session history (ASH) information for a specified duration of a workload that was captured or replayed. Before running this report, AWR data must have been previously exported from the captured or replayed workload. To run this report, under ASH Report, select the workload capture or replay for which you want to generate an ASH report. Specify the duration using the Start Date, Start Time, End Date, and End Time fields. You can also apply filters using the Filter field. Once the duration and filters are specified, click **Run Report**. If AWR data is not previously exported from the captured or replayed workload, you will be prompted to import the AWR data before continuing. For more information about the ASH report, see "Generating Active Session History Reports" on page 5-28.

   The Report window opens while the report is being generated. Once the report is generated, you can save the report by clicking **Save to File**.
10. To return to the Database Replay page, click **OK**.

Replaying a Database Workload Using APIs

This section describes how to replay a database workload using the DBMS_WORKLOAD_REPLAY package. Replaying a database workload using the DBMS_WORKLOAD_REPLAY package is a multi-step process that involves:

- Initializing Replay Data
- Remapping Connections
- Setting Workload Replay Options
- Starting a Workload Replay
- Stopping a Workload Replay
- Exporting AWR Data for Workload Replay

See Also: *Oracle Database PL/SQL Packages and Types Reference*

Initializing Replay Data

After the workload capture is preprocessed and the test system is properly prepared, the replay data can be initialized. Initializing replay data loads the necessary metadata into tables required by workload replay. For example, captured connection strings are loaded into a table where they can be remapped for replay. For information about preprocessing a workload capture, see "Preprocessing a Database Workload" on page 22-13. For information about preparing the test system, see "Setting Up the Test System" on page 22-16.

To initialize replay data, use the INITIALIZE_REPLAY procedure:

```
BEGIN
  DBMS_WORKLOAD_REPLAY.INITIALIZE_REPLAY (replay_name => 'dec06_102',
                          replay_dir => 'dec06');
END;
/
```

In this example, the INITIALIZE_REPLAY procedure loads preprocessed workload data from the dec06 directory into the database.

The INITIALIZE_REPLAY procedure in this example uses the following parameters:

- The replay_name required parameter specifies a replay name that can be used with other APIs to retrieve settings and filters of previous replays.
- The replay_dir required parameter specifies the directory that contains the workload capture that needs to be replayed.

Remapping Connections

After the replay data is initialized, connection strings used in the workload capture need to be remapped so that user sessions can connect to the appropriate databases and perform external interactions as captured during replay. To view connection mappings, use the DBA_WORKLOAD_CONNECTION_MAP view. For information about connection remapping, see "Remapping Connections" on page 22-17.

To remap connections, use the REMAP_CONNECTION procedure:

```
BEGIN
  DBMS_WORKLOAD_REPLAY.REMAP_CONNECTION (connection_id => 101,
                          replay_connection => 'dlsun244:3434/bjava21');
END;
/
```

In this example, the connection that corresponds to the connection ID 101 will use the new connection string defined by the `replay_connection` parameter.

The `REMAP_CONNECTION` procedure in this example uses the following parameters:

- The `connection_id` required parameter is generated when initializing replay data and corresponds to a connection from the workload capture.
- The `replay_connection` optional parameter specifies the new connection string that will be used during workload replay.

Setting Workload Replay Options

After the replay data is initialized and the connections are appropriately remapped, you need to prepare the database for workload replay. For information about workload replay preparation, see "Steps for Replaying a Database Workload" on page 22-16.

To prepare workload replay on the replay system, use the `PREPARE_REPLAY` procedure:

```
BEGIN
  DBMS_WORKLOAD_REPLAY.PREPARE_REPLAY (synchronization => TRUE);
END;
/
```

In this example, the `PREPARE_REPLAY` procedure prepares a replay that has been previously initialized. The `COMMIT` order in the workoad capture will be preserved.

The `PREPARE_REPLAY` procedure uses the following parameters:

- The `synchronization` required parameter determines if synchronization will be used during workload replay. If this parameter is set to `TRUE`, the `COMMIT` order in the captured workload will be preserved during replay and all replay actions will be executed only after all dependent `COMMIT` actions have completed. The default value is `TRUE`.
- The `connect_time_scale` optional parameter scales the elapsed time from when the workload capture started to when the session connects with the specified value and is interpreted as a % value. Use this parameter to increase or decrease the number of concurrent users during replay. The default value is 100.
- The `think_time_scale` optional parameter scales the elapsed time between two successive user calls from the same session and is interpreted as a % value. Setting this parameter to 0 will send user calls to the database as fast as possible during replay. The default value is 100.
- The `think_time_auto_correct` optional parameter corrects the think time (based on the `think_time_scale` parameter) between calls when user calls take longer to complete during replay than during capture and can be set to either `TRUE` or `FALSE`. The default value is `TRUE`.

For more information about setting these parameters, see "Specifying Replay Options" on page 22-17.

Starting a Workload Replay

Before starting a workload replay, you must first:

- Preprocess the captured workload, as described in "Preprocessing a Captured Workload Using APIs" on page 22-15

- Initialize the replay data, as described in "Initializing Replay Data" on page 22-27
- Set up the replay options, as described in "Setting Workload Replay Options" on page 22-28
- Start the replay clients, as described in "Setting Up Replay Clients" on page 22-18

To start a workload replay, use the `START_REPLAY` procedure:

```
BEGIN
  DBMS_WORKLOAD_REPLAY.START_REPLAY ();
END;
/
```

Stopping a Workload Replay

To stop a workload replay, use the `CANCEL_REPLAY` procedure:

```
BEGIN
  DBMS_WORKLOAD_REPLAY.CANCEL_REPLAY ();
END;
/
```

Exporting AWR Data for Workload Replay

Exporting AWR data enables detailed analysis of the workload. This data is also required if you plan to run the AWR Compare Period report on a pair of workload captures or replays.

To export AWR data, use the `EXPORT_AWR` procedure:

```
BEGIN
  DBMS_WORKLOAD_REPLAY.EXPORT_AWR (replay_id => 1);
END;
/
```

In this example, the AWR snapshots that correspond to the workload replay with a replay ID of 1 are exported. The `EXPORT_AWR` procedure uses the `replay_id` required parameter, which specifies the ID of the replay whose AWR snapshots will be exported. This procedure will work only if the corresponding workload replay was performed in the current database and the AWR snapshots that correspond to the original replay time period are still available.

Monitoring Workload Replay Using Views

This section summarizes the views that you can display to monitor workload replay. You need DBA privileges to access these views.

- The `DBA_WORKLOAD_CAPTURES` view lists all the workload captures that have been captured in the current database.
- The `DBA_WORKLOAD_FILTERS` view lists all workload filters, for both workload captures and workload replays, defined in the current database.
- The `DBA_WORKLOAD_REPLAYS` view lists all the workload replays that have been replayed in the current database.
- The `DBA_WORKLOAD_REPLAY_DIVERGENCE` view enables you to monitor workload replay divergence.

- The DBA_WORKLOAD_CONNECTION_MAP view lists the connection mapping information for workload replay.
- The V$WORKLOAD_REPLAY_THREAD view lists information about all sessions from the replay clients.

See Also: *Oracle Database Reference* for information on these views

Analyzing Replayed Workload

This section describes how to generate and analyze workload capture and workload replay reports. The primary tool for generating Database Replay reports is Oracle Enterprise Manager. If for some reason Oracle Enterprise Manager is unavailable, you can generate Database Replay reports using APIs.

This section contains the following topics:

Generating a Workload Capture Report Using Enterprise Manager

The workload capture report contains captured workload statistics, information about the top session activities that were captured, and any workload filters used during the capture process.

To generate a workload capture report using Enterprise Manager:

1. On the Software and Support page, under Real Application Testing, click **Database Replay**.

   The Database Replay page appears.

2. Click **View Workload Capture History**.

   The View Workload Capture History page appears.

3. Select the workload capture for which you want to run a workload capture report and click **View**.

   The View Workload Capture page appears.

4. To view the workload capture report, click **View Workload Capture Report**.

   The Report window opens while the report is being generated.

5. Once the report is generated, you can save the report by clicking **Save to File**.

   For information about how to use a workload capture report, see "Using a Workload Capture Report" on page 22-31.

Generating a Workload Capture Report Using APIs

The workload capture report contains captured workload statistics, information about the top session activities that were captured, and any workload filters used during the capture process.

To generate a report on the latest workload capture, use the DBMS_WORKLOAD_CAPTURE.GET_CAPTURE_INFO procedure and the DBMS_WORKLOAD_CAPTURE.REPORT function:

```
DECLARE
  cap_id         NUMBER;
  cap_rpt        CLOB;
BEGIN
  cap_id  := DBMS_WORKLOAD_CAPTURE.GET_CAPTURE_INFO(dir => 'dec06');
  cap_rpt := DBMS_WORKLOAD_CAPTURE.REPORT(capture_id => cap_id,
                          format => DBMS_WORKLOAD_CAPTURE.TYPE_TEXT);
END;
/
```

In this example, the GET_CAPTURE_INFO procedure retrieves all information regarding the workload capture in the dec06 directory and returns the appropriate cap_id for the workload capture. The REPORT function generates a text report using the cap_id that was returned by the GET_CAPTURE_INFO procedure.

The GET_CAPTURE_INFO procedure uses the dir required parameter, which specifies the name of the workload capture directory object.

The REPORT function uses the following parameters:

- The capture_id required parameter relates to the directory that contains the workload capture for which the report will be generated. The directory should be a valid directory in the host system containing the workload capture. The value of this parameter should match the cap_id returned by the GET_CAPTURE_INFO procedure.
- The format parameter required parameter specifies the report format. Valid values include DBMS_WORKLOAD_CAPTURE.TYPE_TEXT and DBMS_WORKLOAD_REPLAY.TYPE_HTML.

For information about how to use a workload capture report, see "Using a Workload Capture Report" on page 22-31.

See Also: *Oracle Database PL/SQL Packages and Types Reference*

Using a Workload Capture Report

The workload capture report contains various types of information that can be used to assess the validity of the workload capture. Using the information provided in this report, you can determine if the captured workload:

- Represents the actual workload you want to replay
- Does not contain any workload you want to exclude
- Can be replayed

The information contained in the workload capture report are divided into the following categories:

- Details about the workload capture (such as the name of the workload capture, defined filters, date, time, and SCN of capture)
- Overall statistics about the workload capture (such as the total DB time captured, and the number of logins and transactions captured) and the corresponding percentages with respect to total system activity
- Profile of the captured workload

- Profile of the workload that was not captured due to version limitations
- Profile of the uncaptured workload that were excluded using defined filters
- Profile of the uncaptured workload that consists of background process or scheduled jobs

Generating a Workload Replay Report Using Enterprise Manager

The workload replay report contains information that can be used to measure performance differences between the capture system and the replay system.

To generate a workload replay report using Enterprise Manager:

1. On the Software and Support page, under Real Application Testing, click **Database Replay**.

   The Database Replay page appears.

2. In the Go to Task column, click the icon that corresponds to the Replay Workload task.

   The Replay Workload page appears.

3. In the Directory Object list, select a directory that contains the preprocessed workload that was used for the replay for which you want to generate a workload replay report.

   After a directory is selected, the Replay Workload page will be refreshed to display the Capture Summary and the Replay History sections.

4. Under Replay History, select the replay for which you want to generate a workload replay report and click **View**.

   The View Workload Replay page appears.

5. Click **View Workload Replay Report**.

   For information about using the Workload Replay report, see "Using a Workload Replay Report" on page 22-33.

Generating a Workload Replay Report Using APIs

The workload replay report contains information that can be used to measure data and performance divergence between the capture system and the replay system.

To generate a report on the latest workload replay for a workload capture, use the `DBMS_WORKLOAD_REPLAY`.GET_REPLAY_INFO procedure and the `DBMS_WORKLOAD_REPLAY.REPORT` function:

To generate a workload replay report, use the `REPORT` function:

```
DECLARE
  cap_id         NUMBER;
  rep_id         NUMBER;
  rep_rpt        CLOB;
BEGIN
  cap_id := DBMS_WORKLOAD_REPLAY.GET_REPLAY_INFO(dir => 'dec06');
  /* Get the latest replay for that capture */
  SELECT max(id)
  INTO   rep_id
  FROM   dba_workload_replays
  WHERE  capture_id = cap_id;
```

```
  rep_rpt := DBMS_WORKLOAD_REPLAY.REPORT(replay_id => rep_id,
                           format => DBMS_WORKLOAD_REPLAY.TYPE_TEXT);
END;
/
```

In this example, the `GET_REPLAY_INFO` procedure retrieves all information regarding the workload capture in the dec06 directory and the history of all the workload replay attempts from this directory. The procedure first imports a row into `DBA_WORKLOAD_CAPTURES`, which contains information about the workload capture. It then imports a row for every replay attempt retrieved from the replay directory into the `DBA_WORKLOAD_REPLAYS` view. The `SELECT` statement returns the appropriate `rep_id` for the latest replay of the workload. The `REPORT` function generates a text report using the `rep_id` that was returned by the `SELECT` statement.

The `GET_CAPTURE_INFO` procedure uses the `dir` required parameter, which specifies the name of the workload replay directory object.

The `REPORT` function uses the following parameters:

- The `replay_id` required parameter relates to the directory that contains the workload replay for which the report will be generated. The directory should be a valid directory in the host system containing the workload replay. The value of this parameter should match the rep_id returned by the `GET_CAPTURE_INFO` procedure.
- The `format` parameter required parameter specifies the report format. Valid values include DBMS_WORKLOAD_REPLAY.TYPE_TEXT, DBMS_WORKLOAD_REPLAY.TYPE_HTML, and DBMS_WORKLOAD_REPLAY.TYPE_XML.

For information about using the Workload Replay report, see "Using a Workload Replay Report" on page 22-33.

See Also: *Oracle Database PL/SQL Packages and Types Reference*

Using a Workload Replay Report

After the workload is replayed on a test system, there may be some divergence in what is replayed compared to what was captured. There are numerous factors that can cause replay divergence, which can be analyzed using the workload replay report. The information contained in the workload replay report consists of performance and data divergence.

Performance divergence may result when new algorithms are introduced in the replay system that affect the overall performance of the database. For example, if the workload is replayed on a newer version of Oracle Database, a new algorithm may cause specific requests to run faster, and the divergence will appear as a faster execution. In this case, this is a desirable divergence.

Data divergence occurs when the results of DML or SQL queries do not match results that were originally captured in the workload. For example, a SQL statement may return fewer rows during replay than those returned during capture.

Error divergence occurs when the same errors are not encountered during replay and capture.

The information contained in the workload replay report are divided into the following categories:

- Details about the workload replay and the workload capture, such as job name, status, database information, duration and time of each process, and the directory object and path
- Replay options selected for the workload replay and the number of replay clients that were started
- Overall statistics about the workload replay and the workload capture (such as the total DB time captured and replayed, and the number of logins and transactions captured and replay) and the corresponding percentages with respect to total system activity
- Profile of the replayed workload
- Replay divergence
- Error divergence
- DML and SQL query data divergence

23

SQL Performance Analyzer

System changes that affect SQL execution plans, such as upgrading a database or adding new indexes, can severely impact SQL performance. As a result, DBAs spend considerable time identifying and fixing SQL statements that have regressed due to a change.

SQL Performance Analyzer automates the process of assessing the overall effect of a change on the full SQL workload by identifying performance divergence for each statement. A report that shows the net impact on the workload performance due to the change is provided. For regressed SQL statements, SQL Performance Analyzer also provides appropriate executions plan details along with tuning recommendations. As a result, DBAs can remedy any negative outcome before their end users are affected and can validate, with significant time and cost savings, that the system change to the production environment will result in net improvement.

You can use SQL Performance Analyzer to analyze the SQL performance impact of any type of system changes. Examples of common system changes for which you can use SQL Performance Analyzer include:

- Database upgrade
- Configuration changes to the operating system, hardware, or database
- Database initialization parameter changes
- Schema changes, for example, adding new indexes or materialized views
- Gathering optimizer statistics
- SQL tuning actions, for example, creating SQL profiles

This chapter contains the following sections:

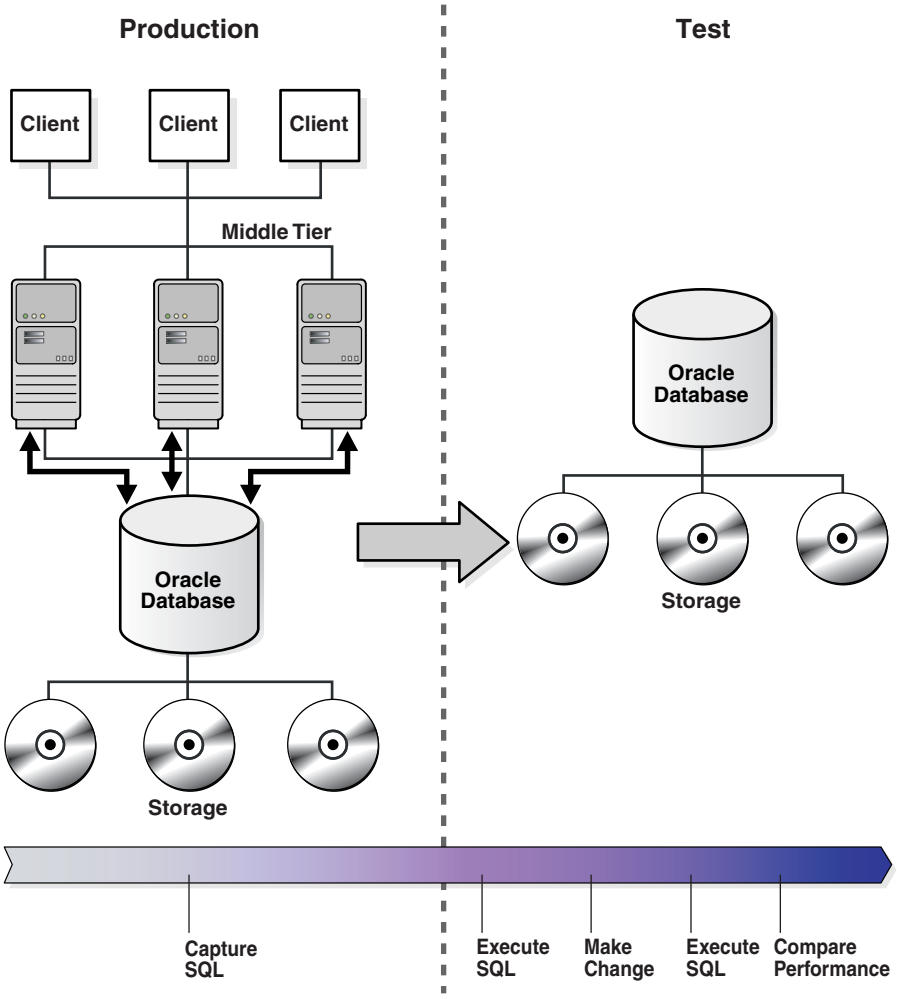
See Also:

- *Oracle Database 2 Day + Performance Tuning Guide* to learn how to run SQL Performance Analyzer with Oracle Enterprise Manager (Enterprise Manager)
- *Oracle Database PL/SQL Packages and Types Reference* for information about the security model for the `DBMS_SQLPA` package

Overview of SQL Performance Analyzer

As illustrated in Figure 23–1, SQL Performance Analyzer evaluates the impact of system changes on SQL performance through five main steps.

Figure 23–1 SQL Performance Analyzer Workflow



The steps of the SQL Performance Analyzer workflow are as follows:

1. Capture the SQL workload.

   You must first capture the set of SQL statements that represents the typical SQL workload on your production system in a SQL Tuning Set (STS). Later, you can conduct the SQL Performance Analyzer analysis on the same database where the workload was captured or on a different database. Because the analysis is resource-intensive, you would typically capture the workload on a production database and perform the analysis on a test database that closely resembles the production system. For more details about how to perform these actions, see "Capturing the SQL Workload" on page 23-5.

2. Measure the performance of the workload before the change.

   SQL Performance Analyzer executes the SQL statements captured in the SQL Tuning Set and generates execution plans and execution statistics for each statement. Only queries and the query part of DML statements are executed to

avoid any side effect on the database. SQL Performance Analyzer executes SQL statements sequentially and in isolation from each other without any respect to their initial order of execution and concurrency. However, you can customize the order in which SQL Performance Analyzer executes the SQL queries. For example, you can start with the most expensive SQL statements in terms of response time.

3. Make a change.

   Make the change whose effect on SQL performance you intend to measure. SQL Performance Analyzer can analyze the effect of many types of system changes. For example, you can test a database upgrade, new index creation, initialization parameter changes, optimizer statistics refresh, and so on.

4. Measure the performance of the workload after the change.

   After you have made the planned change, SQL Performance Analyzer re-executes the SQL statements and produces execution plans and execution statistics for each SQL statement a second time. This execution result represents a new set of performance data that SQL Performance Analyzer uses for subsequent comparison.

5. Compare performance.

   SQL Performance Analyzer compares the performance of SQL statements before and after the change and produces a report identifying any changes in execution plans or performance of the SQL statements.

   If the performance comparison reveals regressed SQL statements, then you can make further changes to remedy the problem. For example, you can fix regressed SQL by running SQL Tuning Advisor or using SQL plan baselines. You can then repeat the process of executing the SQL Tuning Set and comparing its performance to the first execution. Repeat these steps until you are satisfied with the outcome of the analysis.

See Also:

- Chapter 17, "Automatic SQL Tuning" to learn about the SQL Tuning Advisor
- Chapter 15, "Using SQL Plan Management" to learn about SQL plan baselines

Using SQL Performance Analyzer

The primary interface for running SQL Performance Analyzer is Enterprise Manager, as described in *Oracle Database 2 Day + Performance Tuning Guide*. You can also run SQL Performance Analyzer using the `DBMS_SQLPA` PL/SQL package. This section explains how to run SQL performance Analyzer using the main PL/SQL APIs of this package. These APIs are listed in Table 23–1. For more details, see *Oracle Database PL/SQL Packages and Types Reference*.

Table 23–1 Main Functions of the DBMS_SQLPA Package

Function	Purpose
`CREATE_ANALYSIS_TASK`	Create a SQL Performance Analyzer task to process and analyze a SQL Tuning Set
`SET_ANALYSIS_TASK_PARAMETER`	Set a SQL analysis task parameter value
`EXECUTE_ANALYSIS_TASK`	Run a previously created task
`DROP_ANALYSIS_TASK`	Drop a task, deleting all performance data

Table 23–1 (Cont.) Main Functions of the DBMS_SQLPA Package

Function	Purpose
REPORT_ANALYSIS_TASK	Generate a report of analysis task results

To use SQL Performance Analyzer:

1. Capture the SQL workload that you intend to analyze and store it in a SQL Tuning Set.

   "Capturing the SQL Workload" on page 23-5 explains how to perform this task.

2. If you plan to use a test system separate from your production system, then perform the following steps:

   a. Set up the test system so that it matches the production environment as closely as possible.

      "Setting Up the Test System" on page 23-5 explains how to perform this task.

   b. Transport the SQL Tuning Set to the test system.

      "Transporting the Workload to the Test System" on page 23-6 explains how to perform this task.

3. Create a SQL Performance Analyzer task on the test system using the SQL Tuning Set as its input source.

   "Creating a SQL Performance Analyzer Task" on page 23-6 explains how to perform this task.

4. Use SQL Performance Analyzer to build the pre-change performance data by executing the SQL statements stored in the SQL Tuning Set.

   "Executing the SQL Workload Before a Change" on page 23-6 explains how to perform this task.

5. Make a change on the test system.

   "Making a Change" on page 23-7 explains how to perform this task.

6. Use SQL Performance Analyzer to build the post-change performance data by re-executing the SQL statements in the SQL Tuning Set on the post-change test system.

   "Executing the SQL Workload After a Change" on page 23-7 explains how to perform this task.

7. Compare the SQL performance by performing the following steps:

   a. Use SQL Performance Analyzer to compare and analyze the pre-change and post-change versions of performance data.

   b. Generate a report to identify the SQL statements in the SQL workload that have improved, remained unchanged, or regressed after the system change.

   c. Review the report and interpret the results.

   "Comparing SQL Performance" on page 23-8 explains how to perform this task.

8. Optionally, run SQL Tuning Advisor or use SQL plan baselines to prevent regressions caused by execution plan changes.

   Chapter 17, "Automatic SQL Tuning" explains how to tune SQL statements to improve their performance.

Chapter 15, "Using SQL Plan Management" explains how to use SQL Plan Management to avoid performance regressions due to execution plan changes.

9. Ensure that the performance of the tuned SQL statements is acceptable by repeating steps 5 through 8 until your performance goals are met.

Capturing the SQL Workload

You must capture a representative set of SQL statements and store them in a SQL Tuning Set. A SQL Tuning Set is a database object that is used to manage SQL workloads. The STS can be used to store one or more SQL statements along with their execution context, including the text of the SQL, parsing schema under which the SQL statement can be compiled, bind values needed to execute the SQL statement, execution plan, number of times the SQL statement was executed, and so on.

You can load SQL statements into a SQL Tuning Set from different sources, including the cursor cache, Automatic Workload Repository (AWR), and existing SQL Tuning Sets. For more information, see "Loading a SQL Tuning Set" on page 17-16.

To capture the SQL workload, perform one of the following tasks:

- If you do not have a SQL Tuning Set to use as an input for SQL Performance Analyzer, then create an STS as described in "Creating a SQL Tuning Set" on page 17-16.
- If you plan to perform the performance analysis on a test database, then proceed to "Setting Up the Test System" on page 23-5.
- If you plan to perform the performance analysis on the production database only, then skip to "Creating a SQL Performance Analyzer Task" on page 23-6.

Setting Up the Test System

You can run SQL Performance Analyzer on the production database or a test database. If you use a test database, then configure the test database environment to match the database environment of the production system as closely as possible. In this way, SQL Performance Analyzer can more accurately forecast the effect of the system change on SQL performance.

To set up the test system:

1. Create a new database for testing SQL performance.

   There are many ways to create a test database. For example, you can use the `DUPLICATE` command of Recovery Manager (RMAN), Oracle Data Pump, or transportable tablespaces. For best results, the test database should be as similar to the production system as possible. Oracle recommends using RMAN because it can create the test database from pre-existing backups or from the active production datafiles. The production and test databases can reside on the same host or on different hosts.

2. Configure the database environment.

   For example, to test how changing a database initialization parameter will affect SQL performance, set the database initialization parameter on the test system to the same value as the production system. To test a database upgrade to Oracle Database 11*g*, install Oracle Database 11*g* on the test system and revert the `OPTIMIZER_FEATURES_ENABLE` initialization parameter to the database version of the production system.

See Also: *Oracle Database Backup and Recovery User's Guide* to learn how to duplicate a database with RMAN

Transporting the Workload to the Test System

After you have created a SQL Tuning Set with the appropriate SQL workload, export the STS from the production system and import it into the test system where the system change under consideration will be tested.

To transport the SQL workload to the test system, export the SQL Tuning Set from one database to a staging table, then import the set from the staging table into the test database, as described in "Transporting a SQL Tuning Set" on page 17-17.

Creating a SQL Performance Analyzer Task

This section describes how to create a new SQL Performance Analyzer task by using the `DBMS_SQLPA.CREATE_ANALYSIS_TASK` function. A task is a database container for SQL Performance Analyzer execution inputs and results.

Before creating the task, make sure that the SQL workload to use for the performance analysis is available on the system in the form of a SQL Tuning Set. Call the `CREATE_ANALYSIS_TASK` function to prepare the analysis of a SQL Tuning Set using the following parameters:

- Set `task_name` to specify an optional name for the SQL Performance Analyzer task.
- Set `sqlset_name` to the name of the SQL Tuning Set.
- Set `sqlset_owner` to the owner of the SQL Tuning Set. The default is the current schema owner.
- Set `basic_filter` to the SQL predicate used to filter the SQL from the SQL Tuning Set.
- Set `order_by` to specify an order-by clause on the selected SQL.
- Set `top_sql` to consider only the top number of SQL statements after filtering and ranking.

The following example illustrates a function call:

```
VARIABLE t_name VARCHAR2(100);
EXEC :t_name := DBMS_SQLPA.CREATE_ANALYSIS_TASK(sqlset_name => 'my_sts', -
       task_name => 'my_spa_task');
```

See Also: *Oracle Database PL/SQL Packages and Types Reference* to learn more about the `DBMS_SQLPA.CREATE_ANALYSIS_TASK` function

Executing the SQL Workload Before a Change

As explained in "Overview of SQL Performance Analyzer" on page 23-2, you must execute the SQL workload to build the pre-change performance data.

Make sure that you have created a SQL Performance Analyzer task. Call the `EXECUTE_ANALYSIS_TASK` procedure using the following parameters:

- Set the `task_name` parameter to the name of the SQL Performance Analyzer task that you want to execute.
- Set the `execution_type` parameter in either of the following ways:

  - Set to `EXPLAIN PLAN` to generate execution plans for all SQL statements in the SQL Tuning Set without executing them.
  - Set to `TEST EXECUTE` (recommended) to execute all statements in the SQL Tuning Set and generate their execution plans and statistics. When `TEST EXECUTE` is specified, the procedure generates execution plans and execution statistics. The execution statistics enable SQL Performance Analyzer to identify SQL statements that have improved or regressed. Collecting execution statistics in addition to generating execution plans provides greater accuracy in the performance analysis, but takes longer.

- Specify a name to identify the execution using the `execution_name` parameter. If not specified, then SQL Performance Analyzer automatically generates a name for the task execution.
- Specify execution parameters using the `execution_params` parameters. The `execution_params` parameters are specified as (*name*, *value*) pairs for the specified execution. For example, you can set the following execution parameters:
  - The `time_limit` parameter specifies the global time limit to process all SQL statements in a SQL Tuning Set before timing out.
  - The `local_time_limit` parameter specifies the time limit to process each SQL statement in a SQL Tuning Set before timing out.

The following example illustrates a function call made *before* a system change:

```
EXEC DBMS_SQLPA.EXECUTE_ANALYSIS_TASK(task_name => 'my_spa_task', -
       execution_type => 'TEST EXECUTE', -
       execution_name => 'my_exec_BEFORE_change');
```

> **See Also:** *Oracle Database PL/SQL Packages and Types Reference* to learn about the `DBMS_SQLPA.EXECUTE_ANALYSIS_TASK` function

Making a Change

"Overview of SQL Performance Analyzer" on page 23-2 lists examples of possible system changes. For example, you may want to determine how a database initialization parameter change or database upgrade will affect SQL performance. You may also decide to change the system based on recommendations from an advisor such as Automatic Database Diagnostic Monitor (ADDM), SQL Tuning Advisor, or SQL Access Advisor.

Make sure that you executed the SQL workload in the initial environment to generate the pre-change performance data, and then make the system change. For example, if you are testing how changing a database initialization parameter will affect SQL performance, then set the database initialization parameter to a new value.

> **See Also:** "Overview of SQL Performance Analyzer" on page 23-2 for examples of the types of system changes that can be analyzed by SQL Performance Analyzer

Executing the SQL Workload After a Change

After you perform the system change, as described in "Making a Change" on page 23-7, execute the SQL workload again to build the after-change performance data of the workload.

Make sure that you have made the change to the system. Call the `EXECUTE_ANALYSIS_TASK` procedure using the parameters described in "Executing

the SQL Workload Before a Change" on page 23-6. Specify a different value for `execution_name`.

The following example illustrates a function call made *after* a system change:

```
EXEC DBMS_SQLPA.EXECUTE_ANALYSIS_TASK(task_name => 'my_spa_task', -
       execution_type => 'TEST EXECUTE', -
       execution_name => 'my_exec_AFTER_change');
```

> **See Also:** *Oracle Database PL/SQL Packages and Types Reference* to learn about the `DBMS_SQLPA.EXECUTE_ANALYSIS_TASK` function

Comparing SQL Performance

After the post-change SQL performance data is built, you can compare the pre-change version of performance data to the post-change version. Run a comparison analysis using the `DBMS_SQLPA.EXECUTE_ANALYSIS_TASK` procedure or function. Later, you can generate a report that shows the results of the comparison and then interpret the results.

To compare the pre-change and post-change SQL performance data:

1. Call the `EXECUTE_ANALYSIS_TASK` procedure or function using the following parameters:
   - Set the `task_name` parameter to the name of the SQL Performance Analyzer task.
   - Set the `execution_type` parameter to `COMPARE PERFORMANCE`. This setting will analyze and compare two versions of SQL performance data.
   - Specify a name to identify the execution using the `execution_name` parameter. If not specified, it will be generated by SQL Performance Analyzer and returned by the function.
   - Specify two versions of SQL performance data using the `execution_params` parameters. The `execution_params` parameters are specified as (*name*, *value*) pairs for the specified execution. Set the execution parameters that are related to comparing and analyzing SQL performance data as follows:
     - Set the `execution_name1` parameter to the name of the first execution (before the system change was made). This value should correspond to the value of the `execution_name` parameter specified in "Executing the SQL Workload Before a Change" on page 23-6.
     - Set the `execution_name2` parameter to the name of the second execution (after the system change was made). This value should correspond to the value of the `execution_name` parameter specified in "Executing the SQL Workload After a Change" on page 23-7 when you executed the SQL workload after the system change. If the caller does not specify the executions, then by default SQL Performance Analyzer will always compare the last two task executions.
     - Set the `comparison_metric` parameter to specify an expression of execution statistics to use in the performance impact analysis. Possible values include the following metrics or any combination of them: `elapsed_time` (default), `cpu_time`, `buffer_gets`, `disk_reads`, `direct_writes`, and `optimizer_cost`.

For other possible parameters that you can set for comparison, see the description of the DBMS_SQLPA package in *Oracle Database PL/SQL Packages and Types Reference*.

The following example illustrates a function call:

```
EXEC DBMS_SQLPA.EXECUTE_ANALYSIS_TASK(task_name => 'my_spa_task', -
       execution_type => 'COMPARE PERFORMANCE', -
       execution_name => 'my_exec_compare', -
       execution_params => dbms_advisor.arglist(-
                           'comparison_metric', 'buffer_gets'));
```

2. Call the DBMS_SQLPA.REPORT_ANALYSIS_TASK function using the following parameters:

   - Set the task_name parameter to the name of the SQL Performance Analyzer task.
   - Set the execution_name parameter to the name of the execution to use. This value should match the execution_name parameter of the execution for which you want to generate a report.

     To generate a report to display the results of:

     – Execution plans generated for the SQL workload, set this value to match the execution_name parameter of the desired EXPLAIN PLAN execution.

     – Execution plans and execution statistics generated for the SQL workload, set this parameter to match the value of the execution_name parameter used in the desired TEST EXECUTION execution.

     – A comparison analysis, set this value to match the execution_name parameter of the desired ANALYZE PERFORMANCE execution.

     If unspecified, SQL Performance Analyzer generates a report for the last execution.

   - Set the type parameter to specify the type of report to generate. Possible values include TEXT (default), HTML, and XML.
   - Set the level parameter to specify the format of the recommendations. Possible values include TYPICAL (default), BASIC, and ALL.
   - Set the section parameter to specify a particular section to generate in the report. Possible values include SUMMARY (default) and ALL.
   - Set the top_sql parameter to specify the number of SQL statements in a SQL Tuning Set to generate in the report. By default, the report shows the top 100 SQL statements impacted by the system change.

   The following example illustrates a portion of a SQL script that you could use to create and display a comparison summary report:

```
VAR rep   CLOB;
EXEC :rep := DBMS_SQLPA.REPORT_ANALYSIS_TASK('my_spa_task', -
                'text', 'typical', 'summary');
SET LONG 100000 LONGCHUNKSIZE 100000 LINESIZE 130
PRINT :rep
```

3. Review the SQL Performance Analyzer report.

   The report is divided into the following main sections:

   - General Information

- Result Summary
- Result Details

The following sections describe how to review a SQL Performance Analyzer report by using a sample report. This sample report uses `buffer_gets` as the comparison metric to compare the pre-change and post-change executions of a SQL workload.

> **See Also:** *Oracle Database PL/SQL Packages and Types Reference* for information about the `DBMS_SQLPA.EXECUTE_ANALYSIS_TASK` and `DBMS_SQLPA.REPORT_ANALYSIS_TASK` functions

General Information

The General Information section contains basic information and metadata about the SQL Performance Analyzer task, the SQL Tuning Set used, and the pre-change and post-change executions. Example 23–1 shows the General Information section of a sample report.

Example 23–1 General Information

```
-------------------------------------------------------------------------------------------------
General Information
-------------------------------------------------------------------------------------------------

 Task Information:                                Workload Information:
 ---------------------------------------------    ---------------------------------------------
  Task Name    : my_spa_task                       SQL Tuning Set Name        : my_sts
  Task Owner   : APPS                              SQL Tuning Set Owner       : APPS
  Description  :                                   Total SQL Statement Count  : 101

Execution Information:
-------------------------------------------------------------------------------------------------
  Execution Name  : my_exec_compare        Started             : 05/21/2007 11:30:09
  Execution Type  : ANALYZE PERFORMANCE    Last Updated        : 05/21/2007 11:30:10
  Description     :                        Global Time Limit   : UNLIMITED
  Scope           : COMPREHENSIVE          Per-SQL Time Limit  : UNUSED
  Status          : COMPLETED              Number of Errors    : 0

Analysis Information:
-------------------------------------------------------------------------------------------------
 Comparison Metric: BUFFER_GETS
 ------------------
 Workload Impact Threshold: 1%
 --------------------------
 SQL Impact Threshold: 1%
 ----------------------
 Before Change Execution:                         After Change Execution:
 ---------------------------------------------    ---------------------------------------------
  Execution Name      : my_exec_BEFORE_change      Execution Name      : my_exec_AFTER_change
  Execution Type      : TEST EXECUTE               Execution Type      : TEST EXECUTE
  Description         :                            Description         :
  Scope               : COMPREHENSIVE              Scope               : COMPREHENSIVE
  Status              : COMPLETED                  Status              : COMPLETED
  Started             : 05/21/2007 11:22:06        Started             : 05/21/2007 11:25:56
  Last Updated        : 05/21/2007 11:24:01        Last Updated        : 05/21/2007 11:28:30
  Global Time Limit   : 1800                       Global Time Limit   : 1800
  Per-SQL Time Limit  : UNUSED                     Per-SQL Time Limit  : UNUSED
  Number of Errors    : 0                          Number of Errors    : 0

-------------------------------------------------------------------------------------------------
```

In Example 23–1, the Task Information section indicates that the task name is `my_spa_task`. The Workload Information section indicates that the task compares executions of the `my_sts` SQL Tuning Set, which contains 101 SQL statements. As shown in the Execution Information section, the comparison execution is named `my_exec_compare`.

The Analysis Information sections shows that SQL Performance Analyzer compares two executions of the `my_sts` SQL Tuning Set, `my_exec_BEFORE_change` and `my_exec_AFTER_change`, using `buffer_gets` as a comparison metric.

Result Summary

The Result Summary section summarizes the results of the SQL Performance Analyzer task. The Result Summary section is divided into the following subsections:

- Overall Performance Statistics
- Performance Statistics of SQL Statements
- Errors

Overall Performance Statistics The Overall Performance Statistics subsection displays statistics about the overall performance of the entire SQL workload. This section is a very important part of the SQL Performance Analyzer analysis because it shows the impact of the system change on the overall performance of the SQL workload. Use the information in this section to understand the change of the workload performance, and determine whether the workload performance will improve or degrade after making the system change.

Example 23–2 shows the Overall Performance Statistics subsection of a sample report.

Example 23–2 Overall Performance Statistics

```
Report Summary
------------------------------------------------------------------------------------------------

Projected Workload Change Impact:
-------------------------------------------
 Overall Impact      :   47.94%
 Improvement Impact  :   58.02%
 Regression Impact   :  -10.08%

SQL Statement Count
-------------------------------------------
 SQL Category  SQL Count  Plan Change Count
 Overall       101        6
 Improved      2          2
 Regressed     1          1
 Unchanged     98         3
.
.
.
------------------------------------------------------------------------------------------------
```

This example indicates that the overall performance of the SQL workload improved by 47.94%, despite the fact that regressions had a negative impact of -10.08%. After the system change, 2 of the 101 SQL statements ran faster, while 1 ran slower. Performance of 98 statements remained unchanged.

Performance Statistics of SQL Statements The Performance Statistics subsection highlights the SQL statements that are the most impacted by the system change. The pre-change

and post-change performance data for each SQL statement in the workload are compared based on the following criteria:

- Weight, or importance, of each SQL statement
- Impact of the system change on each SQL statement relative to the entire SQL workload
- Impact of the system change on each SQL statement
- Whether the structure of the execution plan for each SQL statement has changed

Example 23–3 shows the Performance Statistics of SQL Statements subsection of a sample report. The report has been altered slightly to fit on the page.

Example 23–3 Performance Statistics of SQL Statements

```
SQL Statements Sorted by their Absolute Value of Change Impact on the Workload
--------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------
|        |               | Impact | Metric  | Metric  | Impact | % Wrkld | % Wrkld | Plan    |
| obj_id | sql_id        | Wrkld  | Before  | After   | on SQL | Before  | After   | Change |
-------------------------------------------------------------------------------------------------
| 205    | 73s2sgy2svfrw |  29.01%| 1681683 |  220590 | 86.88% |   33.39% |  8.42% |   y    |
| 206    | gq2a407mv2hsy |  29.01%| 1681683 |  220590 | 86.88% |   33.39% |  8.42% |   y    |
| 204    | 2wtgxbjz6u2by | -10.08%| 1653012 | 2160529 | -30.7% |   32.82% | 82.48% |   y    |
-------------------------------------------------------------------------------------------------
```

The SQL statements are sorted in descending order by the absolute value of the net impact on the SQL workload, that is, the sort order does not depend on whether the impact was positive or negative.

Errors The Errors subsection reports all errors that occurred during an execution. An error may be reported at the SQL Tuning Set level if it is common to all executions in the SQL Tuning Set, or at the execution level if it is specific to a SQL statement or execution plan.

Example 23–4 shows an example of the Errors subsection of a SQL Performance Analyzer report.

Example 23–4 Errors

```
--------------------------------------------------------------------------------
                            SQL STATEMENTS WITH ERRORS
--------------------------------------------------------------------------------
SQL ID        Error
------------- ------------------------------------------------------------------
47bjmcdtw6htn ORA-00942: table or view does not exist
br61bjp4tnf7y ORA-00920: invalid relational operator
--------------------------------------------------------------------------------
```

Result Details

The Result Details section represents a drill-down into the performance of SQL statements that appears in the Result Summary section of the report. Use the information in this section to investigate why the performance of a particular SQL statement regressed.

This section will contain an entry of every SQL statement processed in the SQL performance impact analysis. Each entry is organized into the following subsections:

- SQL Statement Details

- Single Execution Statistics
- Execution Plans

SQL Statement Details This section of the report summarizes the SQL statement, listing its information and execution details.

Example 23–5 shows the SQL Details subsection of a sample report.

Example 23–5 SQL Details

```
SQL Details:
-----------------------------
 Object ID            : 204
 Schema Name          : APPS
 SQL ID               : 2wtgxbjz6u2by
 Execution Frequency  : 1
 SQL Text             : SELECT /* my_query_14_scott */ /*+ ORDERED INDEX(t1)
                        USE_HASH(t1) */ 'B' || t2.pg_featurevalue_05_id
                        pg_featurevalue_05_id, 'r' || t4.elementrange_id
                        pg_featurevalue_15_id, 'G' || t5.elementgroup_id
                        pg_featurevalue_01_id, 'r' || t6.elementrange_id . . .
.
.
.
--------------------------------------------------------------------------------------------
```

In Example 23–5, the report summarizes the regressed SQL statement whose ID is `2wtgxbjz6u2by` and corresponding object ID is `204`.

Single Execution Statistics The Single Execution Statistics subsection compares execution statistics of the SQL statement from the pre-change and post-change executions and then summarizes the findings.

Example 23–6 shows the Single Execution Statistics subsection of a sample report.

Example 23–6 Single Execution Statistics

```
Execution Statistics:
-----------------------------
-------------------------------------------------------------------------------------------
|              | Impact on | Value    | Value    | Impact      | % Workload | % Workload |
| Stat Name    | Workload  | Before   | After    | on SQL      | Before     | After      |
-------------------------------------------------------------------------------------------
| elapsed_time |    -95.54%|  36.484  | 143.161  |   -292.39%  |     32.68% |     94.73% |
| parse_time   |    -12.37%|    .004  |    .062  |     -1450%  |       .85% |     11.79% |
| exec_elapsed |    -95.89%|   36.48  | 143.099  |   -292.27%  |     32.81% |     95.02% |
| exec_cpu     |    -19.73%|  36.467  |  58.345  |    -59.99%  |     32.89% |     88.58% |
| buffer_gets  |    -10.08%| 1653012  | 2160529  |     -30.7%  |     32.82% |     82.48% |
| cost         |     12.17%|   11224  |    2771  |     75.31%  |     16.16% |      4.66% |
| reads        |  -1825.72%|    4091  |  455280  | -11028.82%  |     16.55% |     96.66% |
| writes       |     -1500%|       0  |      15  |     -1500%  |         0% |       100% |
| rows         |           |     135  |     135  |             |            |            |
-------------------------------------------------------------------------------------------

Findings (2):
-----------------------------
 1. The performance of this SQL has regressed.
 2. The structure of the SQL execution plan has changed.
--------------------------------------------------------------------------------------------
```

Execution Plans The Execution Plans subsection displays the pre-change and post-change execution plans for the SQL statement. In cases when the performance regressed, this section also contains findings on root causes and symptoms.

Example 23–7 shows the Execution Plans subsection of a sample report.

Example 23–7 Execution Plans

```
Execution Plan Before Change:
-----------------------------
 Plan Id          : 1
 Plan Hash Value  : 3412943215

-----------------------------------------------------------------------------------------------------------
| Id   | Operation                          | Name                | Rows   | Bytes    | Cost  | Time     |
-----------------------------------------------------------------------------------------------------------
|    0 | SELECT STATEMENT                   |                     |      1 |      126 | 11224 | 00:02:15 |
|    1 |   HASH GROUP BY                    |                     |      1 |      126 | 11224 | 00:02:15 |
|    2 |    NESTED LOOPS                    |                     |      1 |      126 | 11223 | 00:02:15 |
|  * 3 |     HASH JOIN                      |                     |      1 |      111 | 11175 | 00:02:15 |
|  * 4 |      TABLE ACCESS FULL             | LU_ELEMENTGROUP_REL |      1 |       11 |   162 | 00:00:02 |
|  * 5 |      HASH JOIN                     |                     |    487 |    48700 | 11012 | 00:02:13 |
|    6 |       MERGE JOIN                   |                     |     14 |      924 |  1068 | 00:00:13 |
|    7 |        SORT JOIN                   |                     |   5391 |   274941 |  1033 | 00:00:13 |
|  * 8 |         HASH JOIN                  |                     |   5391 |   274941 |   904 | 00:00:11 |
|  * 9 |          TABLE ACCESS FULL         | LU_ELEMENTGROUP_REL |    123 |     1353 |   175 | 00:00:03 |
| * 10 |          HASH JOIN                 |                     |   5352 |   214080 |   729 | 00:00:09 |
| * 11 |           TABLE ACCESS FULL        | LU_ITEM_293         |   5355 |   128520 |    56 | 00:00:01 |
| * 12 |           TABLE ACCESS FULL        | ADM_PG_FEATUREVALUE |   1629 |    26064 |   649 | 00:00:08 |
| * 13 |        FILTER                      |                     |        |          |       |          |
| * 14 |         SORT JOIN                  |                     |      1 |       15 |    36 | 00:00:01 |
| * 15 |          TABLE ACCESS FULL         | LU_ELEMENTRANGE_REL |      1 |       15 |    35 | 00:00:01 |
|   16 |       INLIST ITERATOR              |                     |        |          |       |          |
| * 17 |        TABLE ACCESS BY INDEX ROWID | FACT_PD_OUT_ITM_293 | 191837 |  6522458 |  9927 | 00:02:00 |
|   18 |         BITMAP CONVERSION TO ROWIDS |                     |        |          |       |          |
| * 19 |          BITMAP INDEX SINGLE VALUE | FACT_274_PER_IDX    |        |          |       |          |
| * 20 |     TABLE ACCESS FULL              | LU_ELEMENTRANGE_REL |      1 |       15 |    49 | 00:00:01 |
-----------------------------------------------------------------------------------------------------------
.
.
.

Execution Plan After Change:
-----------------------------
 Plan Id          : 102
 Plan Hash Value  : 1923145679

-----------------------------------------------------------------------------------------------------
| Id   | Operation                          | Name                | Rows | Bytes    | Cost | Time     |
-----------------------------------------------------------------------------------------------------
|    0 | SELECT STATEMENT                   |                     |    1 |      126 | 2771 | 00:00:34 |
|    1 |   HASH GROUP BY                    |                     |    1 |      126 | 2771 | 00:00:34 |
|    2 |    NESTED LOOPS                    |                     |    1 |      126 | 2770 | 00:00:34 |
|  * 3 |     HASH JOIN                      |                     |    1 |      111 | 2722 | 00:00:33 |
|  * 4 |      HASH JOIN                     |                     |    1 |      100 | 2547 | 00:00:31 |
|  * 5 |       TABLE ACCESS FULL            | LU_ELEMENTGROUP_REL |    1 |       11 |  162 | 00:00:02 |
|    6 |       NESTED LOOPS                 |                     |      |          |      |          |
|    7 |        NESTED LOOPS                |                     |  484 |    43076 | 2384 | 00:00:29 |
|  * 8 |         HASH JOIN                  |                     |   14 |      770 |  741 | 00:00:09 |
|    9 |          NESTED LOOPS              |                     |    4 |      124 |  683 | 00:00:09 |
| * 10 |           TABLE ACCESS FULL        | LU_ELEMENTRANGE_REL |    1 |       15 |   35 | 00:00:01 |
| * 11 |           TABLE ACCESS FULL        | ADM_PG_FEATUREVALUE |    4 |       64 |  649 | 00:00:08 |
| * 12 |          TABLE ACCESS FULL         | LU_ITEM_293         | 5355 |   128520 |   56 | 00:00:01 |
|   13 |         BITMAP CONVERSION TO ROWIDS |                     |      |          |      |          |
| * 14 |          BITMAP INDEX SINGLE VALUE | FACT_274_ITEM_IDX   |      |          |      |          |
| * 15 |        TABLE ACCESS BY INDEX ROWID | FACT_PD_OUT_ITM_293 |   36 |     1224 | 2384 | 00:00:29 |
| * 16 |      TABLE ACCESS FULL             | LU_ELEMENTGROUP_REL |  123 |     1353 |  175 | 00:00:03 |
| * 17 |     TABLE ACCESS FULL              | LU_ELEMENTRANGE_REL |    1 |       15 |   49 | 00:00:01 |
-----------------------------------------------------------------------------------------------------
```

SQL Performance Analyzer Views

You can query the following views to monitor SQL Performance Analyzer and view its analysis results:

- The DBA_ADVISOR_TASKS and USER_ADVISOR_TASKS views display descriptive information about the SQL Performance Analyzer task that was created.
- The DBA_ADVISOR_EXECUTIONS and USER_ADVISOR_EXECUTIONS views display information about task executions. SQL Performance Analyzer creates at least three executions to analyze the SQL performance impact caused by a database change on a SQL workload. The first execution collects a pre-change version of the performance data. The second execution collects a post-change version of the performance data. The third execution performs the comparison analysis.
- The DBA_ADVISOR_FINDINGS and USER_ADVISOR_FINDINGS views display the SQL Performance Analyzer findings. SQL Performance Analyzer generates the following types of findings:
  - Problems, such as performance regression
  - Symptoms, such as when the structure of an execution plan has changed
  - Errors, such as nonexistence of an object or view
  - Informative messages, such as when the structure of an execution plan in the pre-change version is different than the one stored in the SQL Tuning Set
- The DBA_ADVISOR_SQLPLANS and USER_ADVISOR_SQLPLANS views display a list of all execution plans.
- The DBA_ADVISOR_SQLSTATS and USER_ADVISOR_SQLSTATS views display a list of all SQL compilations and execution statistics.

You must have the SELECT_CATALOG_ROLE role to access the DBA views.

> **See Also:** *Oracle Database Reference* for information about the DBA_ADVISOR_TASKS, DBA_ADVISOR_EXECUTIONS, and DBA_ADVISOR_SQLPLANS views

Example: Analyzing SQL Performance Impact of a Database Upgrade

This example describes how to use SQL Performance Analyzer to analyze the SQL performance impact of a database upgrade from Oracle Database 10*g* Release 2 to Oracle Database 11*g* Release 1.

1. On the production system running Oracle Database 10*g*, capture the SQL workload that you want to use in the analysis into a SQL Tuning Set.
2. Export the SQL Tuning Set and import it into the test system running Oracle Database 11*g*.
3. Set up the initial environment on the test system. In this case, set the OPTIMIZER_FEATURES_ENABLE initialization parameter to the database version from which you are upgrading, which is 10.2 in this scenario. You should use this parameter to test a database upgrade because SQL Performance Analyzer does not exist in releases before Oracle Database 11*g*. Thus, you cannot use SQL Performance Analyzer before an upgrade to build the pre-change version of the workload.

4. Create a SQL Performance Analyzer task using the SQL Tuning Set as the input source.
5. Execute the SQL workload to compute the pre-change version of the SQL workload.
6. Set the `OPTIMIZER_FEATURES_ENABLE` initialization parameter to the database version to which you are upgrading.
7. Execute the SQL workload a second time to compute the post-change version of the SQL workload.
8. Perform a comparison analysis to analyze the change in performance between the pre-change and post-change executions.

Review the SQL Performance Analyzer report to identify improvements or degradation to the performance of the SQL workload. In cases where the performance degraded, use the report to identify the SQL statements that have regressed and fix them with SQL Tuning Advisor or SQL plan baselines.

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.

Automatic Workload Repository

Collects, processes, and maintains performance statistics for problem detection and self-tuning purposes.

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.

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.

direct I/O

I/O which bypasses the buffer cache. See "PIO" on page Glossary-4.

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 enables examination of 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 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.

MTBF

Mean time between failures. A common database statistic important to tuning I/O.

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 Query Optimizer.

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 which bypasses the buffer cache.

plan generator

Tries out different possible plans for a given query so that the query optimizer 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 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 query optimizer is made up of the query transformer, the estimator, and the plan generator.

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.

Note: This feature has been desupported.

---

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 Profile

A collection of information that enables the query optimizer to create an optimal execution plan for a SQL statement.

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 Tuning Set (STS)

A database object that includes one or more SQL statements along with their execution statistics and execution context.

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. This feature has been replaced by the Automatic Workload Repository.

striping

The interleaving of a related block of data across disks. Proper striping reduces I/O and improves performance.

- Stripe depth is the size of the stripe, sometimes called stripe unit.
- Stripe width is the product of the stripe depth and the number of drives in the striped set.

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.

work area

A private allocation of memory used for sorts, hash joins, and other operations that are memory-intensive. 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.

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W