Oracle8*i*

Tuning

Release 8.1.5

February 1999

Part No. A67775-01

Oracle8*i* Tuning

Part No. A67775-01

Release 8.1.5

Primary Author: Mark Bauer.

Primary Contributors: Benoit Dageville, Lilian Hobbs, Paul Lane, Lefty Leverenz, Rita Moran, Juan Tellez, Graham Wood, and Mohamed Zait.

Contributors: Tomohiro Akiba, Lance Ashdown, David Austin, Andre Bakker, Allen Brumm, Dave Colello, Carol Colrain, Dean Daniels, Dinesh Das, Sohan Demel, Michael Depledge, Joyce Fee, Jyotin Gautam, Jackie Gosselin, Scott Gossett, John Graham, Todd Guay, Gary Hallmark, Mike Hartstein, Scott Heisey, Alex Ho, Andrew Holdsworth, Hakan Jakobssen, Sue Jang, Robert Jenkins, Ashok Joshi, Paul Justus, Jan Klokkers, Anjo Kolk, Dan Leary, Tirthankar Lahiri, Juan Loaiza, Diana Lorentz, George Lumpkin, Roderick Manalac, Sheryl Maring, Ravi Mirchandaney, Ken Morse, Ari Mozes, Jeff Needham, Kotaro Ono, Cetin Ozbutun, Orla Parkinson, Doug Rady, Mary Rhodes, Ray Roccaforte, Hari Sankar, Ekrem Soylemez, Leng Leng Tan, Bob Thome, Lawrence To, Dan Tow, Randy Urbano, Steve Vivian, Sandy Venning, Bill Waddington, and Steve Wertheimer.

Graphic Designer: Valarie Moore.

Send Us Your Comments

Oracle8*i* Tuning, Release 8.1.5

Part No. A67775-01

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

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

If you find errors or have suggestions, please indicate the chapter, section, and page number (if available). Send comments and suggestions to the Information Development department using any of the following:

- Email: infodev@us.oracle.com
- FAX: 650-506-7228. Attn: Oracle8*i* Tuning
- Postal Service:
  Oracle Corporation
  Server Technologies Documentation
  500 Oracle Parkway, 4OP12
  Redwood Shores, CA 94065
  U.S.A.

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

Preface

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

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

For information about the differences between Oracle8*i* and the Oracle8*i* Enterprise Edition and the available features and options, please refer to *Getting to Know Oracle8i*.

Intended Audience

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

How This Book is Organized

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

Part One: Introduction to Tuning

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

Part Two: Application Design Tuning for Designers and Programmers

Chapter 3, "Application and System Performance Characteristics"	This chapter describes the various types of application that use Oracle databases and the suggested approaches and features available when designing each.
Chapter 4, "Tuning Database Operations"	This chapter explains the fundamentals of tuning database operations.
Chapter 5, "Registering Applications"	This chapter describes how to register an application with the database and retrieve statistics on each registered module or code segment.
Chapter 6, "Data Access Methods"	This chapter provides an overview of data access methods that can enhance performance, and warns of situations to avoid.
Chapter 7, "Optimizer Modes, Plan Stability, and Hints"	This chapter explains when to use the available optimization modes and how to use hints to enhance Oracle performance.
Chapter 8, "Tuning Distributed Queries"	This chapter provides guidelines for tuning distributed queries.
Chapter 9, "Transaction Modes"	This chapter describes the different methods in which read consistency is performed.

Part Three: Application Design Tools for Designers and DBAs

Chapter 10, "Managing SQL and Shared PL/SQL Areas"	This chapter explains the use of shared SQL to improve performance.
Chapter 11, "Optimizing Data Warehouse Applications"	This chapter introduces integrated Oracle8*i* features for tuning enterprise-scale data warehouses.
Chapter 12, "Overview of Diagnostic Tools"	This chapter introduces the full range of diagnostic tools available for monitoring production systems and determining performance problems.
Chapter 13, "Using EXPLAIN PLAN"	This chapter shows how to use the SQL command EXPLAIN PLAN, and format its output.
Chapter 14, "The SQL Trace Facility and TKPROF"	This chapter describes the use of the SQL trace facility and TKPROF, two basic performance diagnostic tools that can help you monitor and tune applications that run against the Oracle Server.
Chapter 15, "Using Oracle Trace"	This chapter provides an overview of Oracle Trace usage and describes the Oracle Trace initialization parameters.
Chapter 16, "Dynamic Performance Views"	This chapter describes views that are of the greatest use for both performance tuning and ad hoc investigation

Part Four: Optimizing Oracle Instance Performance

Chapter 17, "Diagnosing System Performance Problems"	This chapter provides an overview of performance factors in existing systems that have been properly designed.
Chapter 18, "Tuning CPU Resources"	This chapter describes how to identify and solve problems with CPU resources.
Chapter 19, "Tuning Memory Allocation"	This chapter explains how to allocate memory to database structures. Proper sizing of these structures can greatly improve database performance.
Chapter 20, "Tuning I/O"	This chapter explains how to avoid I/O bottlenecks that could prevent Oracle from performing at its maximum potential.
Chapter 21, "Tuning Resource Contention"	This chapter explains how to detect and reduce contention that affects performance.
Chapter 22, "Tuning Networks"	This chapter introduces networking issues that affect tuning, and points to the use of array interfaces, out-of-band breaks, and other tuning techniques.

Chapter 23, "Tuning the Multi-Threaded Server Architecture"	This chapter explains how to tune the components of the Multi-threaded Server architecture.
Chapter 24, "Tuning the Operating System"	This chapter explains how to tune the operating system for optimal performance of Oracle.
Chapter 25, "Tuning Instance Recovery Performance"	This chapter explains how to tune recovery performance.

Part Five: Parallel Execution

Chapter 26, "Tuning Parallel Execution"	This chapter explains how to use and tune parallel execution features for improved performance. It also describes how to optimize partitioning.
Chapter 27, "Understanding Parallel Execution Performance Issues"	This chapter provides a conceptual explanation of parallel execution performance issues and explains how to diagnose and solve parallel execution performance problems.

Part Six: Materialized Views

Chapter 28, "Data Warehousing with Materialized Views"	This chapter discusses data warehousing and how to use materialized views to optimize data warehouse operations.
Chapter 29, "Materialized Views"	This chapter provides a general overview of materialized views.
Chapter 30, "Dimensions"	This chapter describes optimizing dimensions within materialized views.
Chapter 31, "Query Rewrite"	This chapter describes how to use query rewrites to optimize your use of materialized views.
Chapter 32, "Managing Materialized Views"	This chapter explains how to manage materialized views.

Related Documents

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

For more information about Oracle Enterprise Manager and its optional applications, please see the following publications:

Oracle Enterprise Manager Concepts Guide

Oracle Enterprise Manager Administrator’s Guide

Oracle Enterprise Manager Application Developer’s Guide

Oracle Enterprise Manager: Introducing Oracle Expert

Oracle Enterprise Manager: Oracle Expert User’s Guide

Oracle Enterprise Manager Performance Monitoring User’s Guide. This manual describes how to use Oracle TopSessions, Oracle Monitor, and Oracle Tablespace Manager.

Conventions

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

- Text
- Syntax diagrams and notation
- Code examples

Text

This section explains the conventions used within the text:

UPPERCASE Characters

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

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

Italicized Characters

Italicized words within text are book titles or emphasized words.

Syntax Diagrams and Notation

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

Keywords

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

Parameters

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

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

Parameter	Description	Examples
table	The substitution value must be the name of an object of the type specified by the parameter.	emp
’text’	The substitution value must be a character literal in single quotes.	’Employee Records’
condition	The substitution value must be a condition that evaluates to TRUE or FALSE.	ename > ’A’
date	The substitution value must be a date constant or an expression of DATE datatype.	TO_DATE (’01-Jan-1996’, DD-MON-YYYY’)
expr	The substitution value can be an expression of any datatype.	sal + 1000
integer	The substitution value must be an integer.	72
rowid	The substitution value must be an expression of datatype ROWID.	00000462.0001.0001
subquery	The substitution value must be a SELECT statement contained in another SQL statement.	SELECT ename FROM emp
statement_name *block_name*	The substitution value must be an identifier for a SQL statement or PL/SQL block.	s1 b1

Code Examples

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

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

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

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

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

Contents

Part I

Introduction to Tuning

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

1 Introduction to Oracle Performance Tuning

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

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

This book begins by describing tuning and explaining tuning methods. Then Part Two describes how system designers and programmers can plan for optimal performance. Part Three explains the design tools for designers and DBAs. Part Four explains how to optimize performance during production. Parts Five and Six describe parallel execution and Materialized Views respectively.

Topics in this chapter include:

What Is Performance Tuning?

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

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

Trade-offs Between Response Time and Throughput

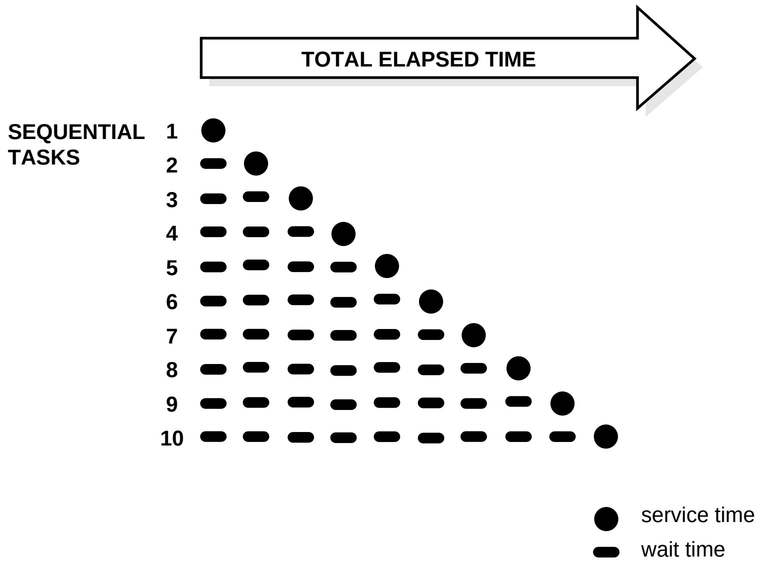
Goals for tuning vary, depending on the needs of the application. Online transaction processing (OLTP) applications define performance in terms of throughput. These applications must process thousands or even millions of very small transactions per day. By contrast, decision support systems (DSS applications)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

Response Time

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

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

Figure 1–1 Sequential Processing of Multiple Independent Tasks



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

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

System Throughput

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

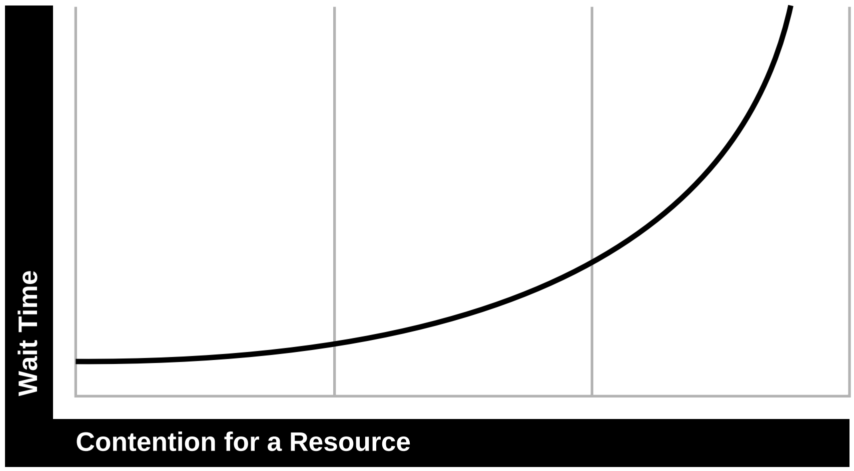
- Get more work done with the same resources (reduce service time).
- Get the work done quicker by reducing overall response time. To do this, look at the wait time. You may be able to duplicate the resource for which all the

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

Wait Time

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

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



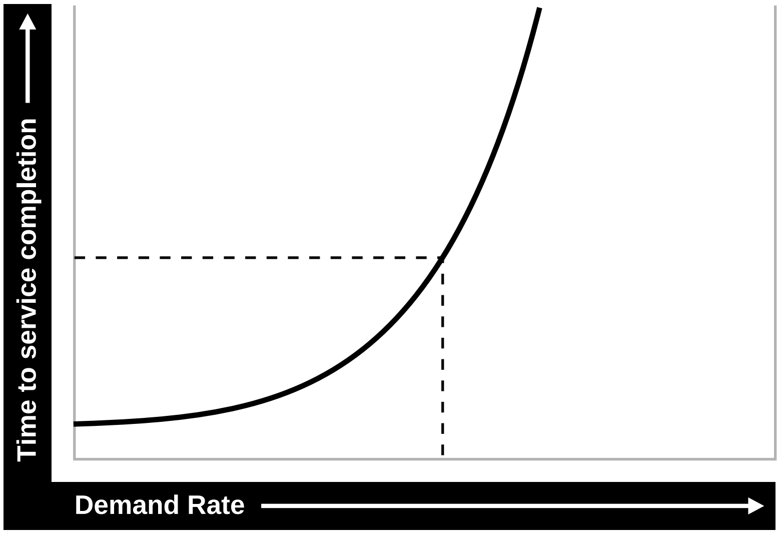
Critical Resources

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

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

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

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



To manage this situation, you have two options:

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

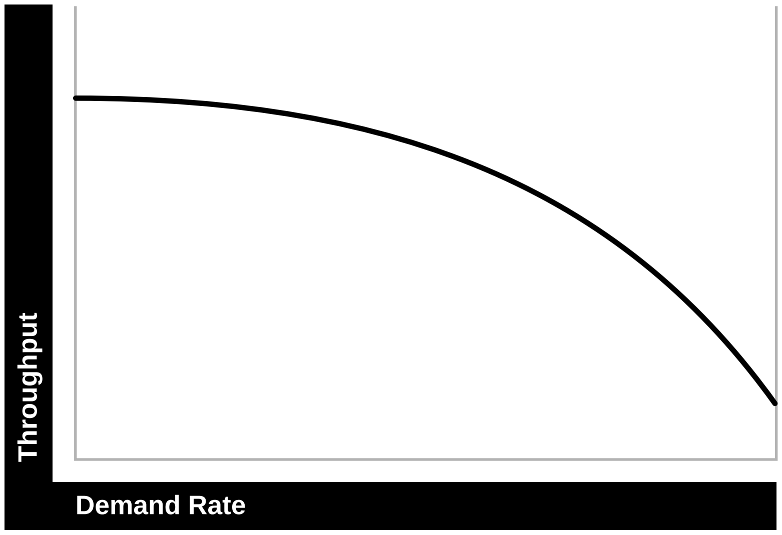
Effects of Excessive Demand

Excessive demand gives rise to:

- Greatly increased response time
- Reduced throughput

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

Figure 1–4 Increased Response Time/Reduced Throughput



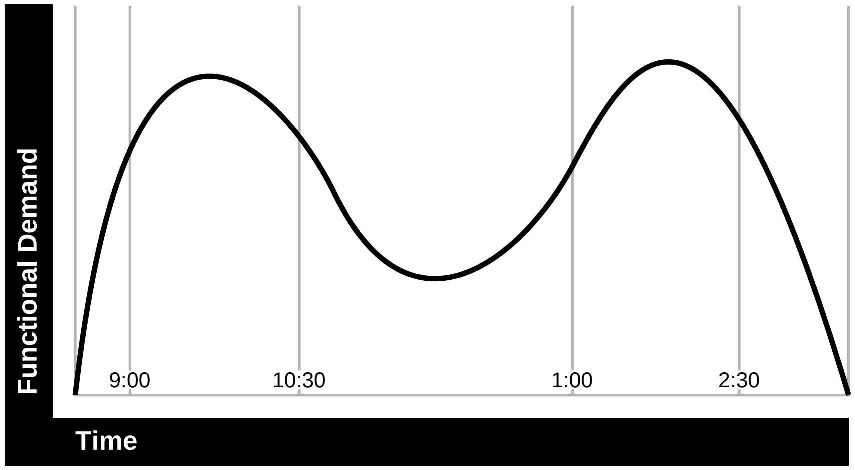
Adjustments to Relieve Problems

You can relieve performance problems by making the following adjustments:

Adjusting unit consumption	You can relieve some problems by using fewer resources per transaction or by reducing service time. Or you can take other approaches, such as reducing the number of I/Os per transaction.
Adjusting functional demand	Other problems can be solved by rescheduling or redistributing the work.
Adjusting capacity	You can also relieve problems by increasing or reallocating resource. If you start using multiple CPUs, going from a single CPU to a symmetric multiprocessor, multiple resources are available.

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

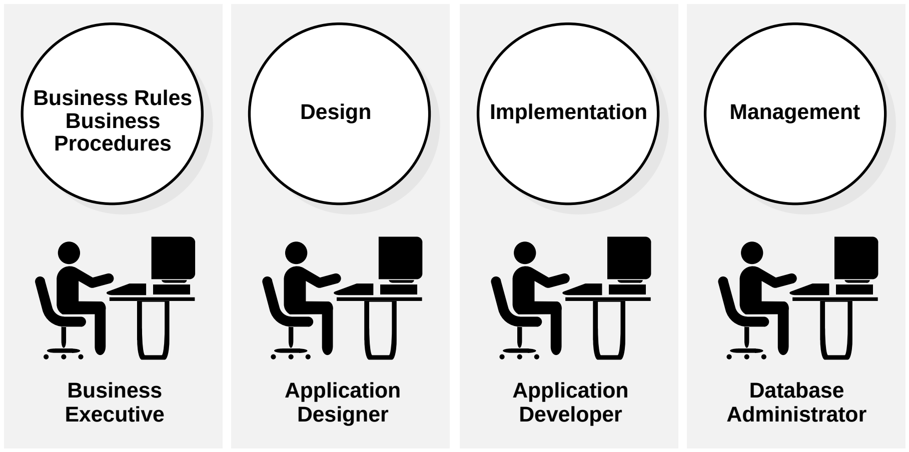
Figure 1–5 Adjusting Capacity and Functional Demand



Who Tunes?

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

Figure 1–6 Who Tunes the System?



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

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

See Also: Chapter 17, "Diagnosing System Performance Problems" for problem-solving methods that can help identify and solve performance problems.

Setting Performance Targets

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

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

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

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

Setting User Expectations

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

If a DBA promises 1-second response time, consider how this might be interpreted. The DBA might mean that the operation would take 1 second in the database—and might well be able to achieve this goal. However, users querying over a network might experience a delay of a couple of seconds due to network traffic: they may not receive the response they expect in 1 second.

Evaluating Performance

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

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

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

See Also: Chapter 12, "Overview of Diagnostic Tools".

2

Performance Tuning Methods

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

Topics in this chapter include:

See Also: "Oracle Expert". Oracle Expert automates the process of collecting and analyzing data. It also provides database tuning recommendations, implementation scripts, and performance reports.

When Is Tuning Most Effective?

For best results, tune during the design phase rather than waiting to tune after implementing your system.

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

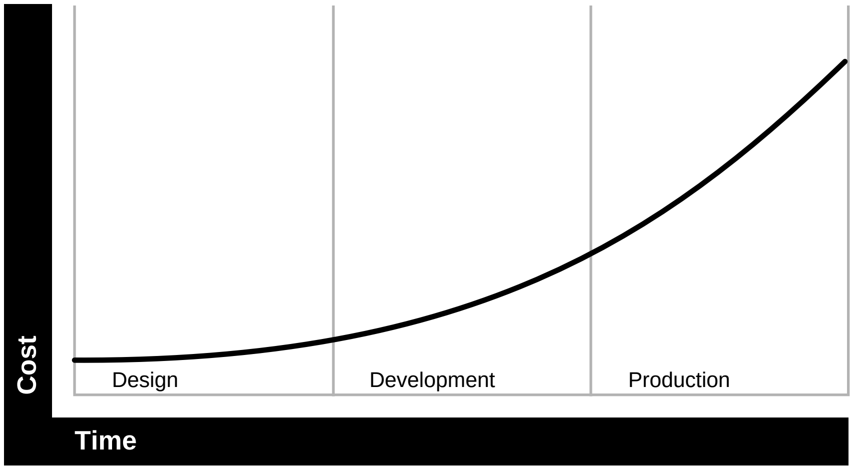
Proactive Tuning While Designing and Developing Systems

By far the most effective approach to tuning is the pro-active approach. Begin by following the steps described in this chapter under "Prioritized Tuning Steps" on page 2-4.

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

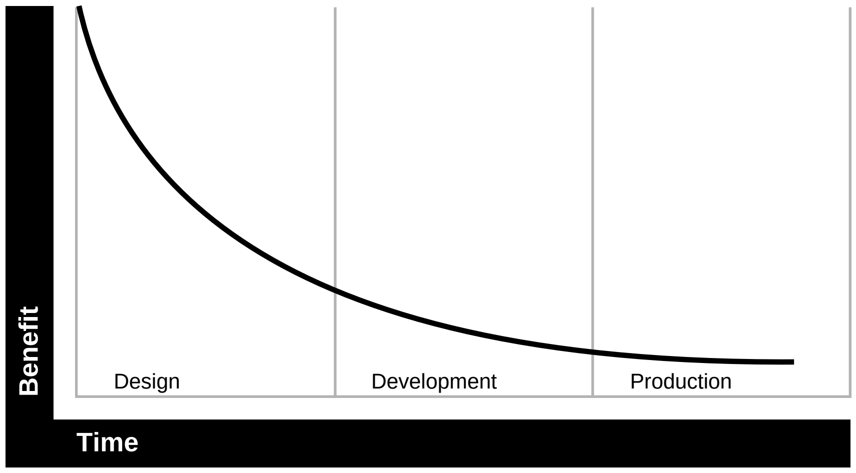
By designing a system to perform well, you can minimize its implementation and on-going administration cost. Figure 2–1 illustrates the relative *cost* of tuning during the life of an application.

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



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

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



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

Reactive Tuning to Improve Production Systems

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

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

Alternatively, a change to the business rule may be necessary to make the system more scalable. If you change the rule so the manager only needs to approve withdrawals exceeding $150, you have created a scalable solution. In this situation, effective tuning could only be done at the highest design level rather than at the end of the process.

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

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

> **See Also:** Part 4: Optimizing Oracle Instance Performance. This describes how to tune CPU, memory, I/O, networks, contention, and the operating system. Also refer to *Oracle8i Concepts* This text provides background on the Oracle server architecture and features so you can locate performance bottlenecks quickly and easily and determine the corrective action.

Prioritized Tuning Steps

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

Step 1: Tune the Business Rules

Step 2: Tune the Data Design

Step 3: Tune the Application Design

Step 4: Tune the Logical Structure of the Database

Step 5: Tune Database Operations

Step 6: Tune the Access Paths

Step 7: Tune Memory Allocation

Step 8: Tune I/O and Physical Structure

Step 9: Tune Resource Contention

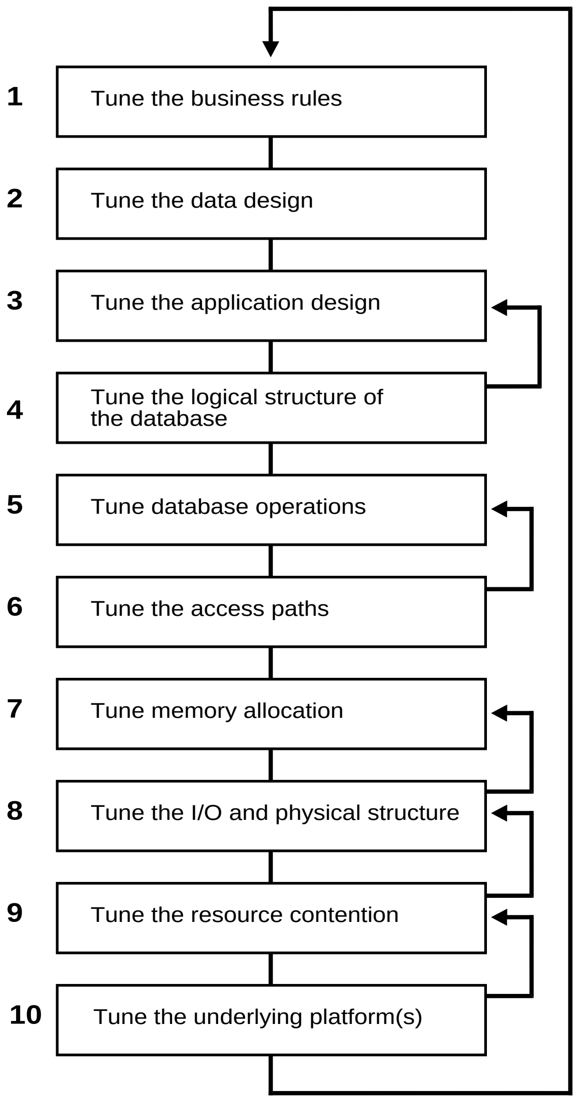
Step 10: Tune the Underlying Platform(s)

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

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

Figure 2–3 illustrates the tuning method:

Figure 2–3 The Tuning Method


1
Tune the business rules
2
Tune the data design
3
Tune the application design
4
Tune the logical structure of the database
5
Tune database operations
6
Tune the access paths
7
Tune memory allocation
8
Tune the I/O and physical structure
9
Tune the resource contention
10
Tune the underlying platform(s)

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

Step 1: Tune the Business Rules

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

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

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

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

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

Step 2: Tune the Data Design

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

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

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

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

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

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

- Consider whether to partition your data
- Consider whether to use local or global indexes

See Also: Chapter 2, "Performance Tuning Methods", "Phase Three - Creating, Populating, and Refreshing the Database" on page 26-63, and *Oracle8i Concepts* for discussions of partitioning and indexes.

Step 3: Tune the Application Design

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

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

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

Step 4: Tune the Logical Structure of the Database

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

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

See Also: "Using Indexes" on page 6-2.

Step 5: Tune Database Operations

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

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

Understanding Oracle’s query processing mechanisms is also important for writing effective SQL statements. Chapter 7, "Optimizer Modes, Plan Stability, and Hints", discusses Oracle’s query optimizer and how to write statements to achieve optimal performance. This chapter also discusses optimizer statistics management and describes preserving execution plans with the plan stability feature.

See Also: Part IV, Optimizing Instance Performance.

Step 6: Tune the Access Paths

Ensure that there is efficient data access. Consider the use of clusters, hash clusters, B*-tree indexes, and bitmap indexes.

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

See Also: Chapter 6, "Data Access Methods".

Step 7: Tune Memory Allocation

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

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

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

You can explicitly set memory allocation for the following structures:

- Buffer cache
- Log buffer
- Sequence caches

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

Process local areas include:

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

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

> **See Also:** Chapter 19, "Tuning Memory Allocation" and *Oracle8i Concepts* for information about memory structures and processes.

Step 8: Tune I/O and Physical Structure

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

- Distributing data so I/O is distributed to avoiding disk contention
- Storing data in data blocks for best access: setting an adequate number of free lists and using proper values for PCTFREE and PCTUSED
- Creating extents large enough for your data so as to avoid dynamic extension of tables; this would adversely effect the performance of high-volume OLTP applications
- Evaluating the use of raw devices

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

Step 9: Tune Resource Contention

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

- Block contention
- Shared pool contention
- Lock contention

- Pinging (in a parallel server environment)
- Latch contention

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

Step 10: Tune the Underlying Platform(s)

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

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

See Also: Chapter 24, "Tuning the Operating System".

Applying the Tuning Method

This section explains how to apply the tuning method:

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

Set Clear Goals for Tuning

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

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

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

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

Create Minimum Repeatable Tests

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

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

Test Hypotheses

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

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

Keep Records and Automate Testing

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

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

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

Avoid Common Errors

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

A good way to scrutinize your resolution process is to develop a written description of your theory of what you think the problem is. This often helps you detect mistakes, simply from articulating your ideas. For best results, consult a team of people to help resolve performance problems. While a performance tuner can tune SQL statements without knowing the application in detail, the team should include someone who understands the application and who can validate the solutions the SQL tuner may devise.

Avoid Poorly Thought Out Solutions

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

Avoid Preconceptions

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

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

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

Stop Tuning When Objectives Are Met

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

Demonstrate Meeting the Objectives

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

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

The next section describes application design tuning for designers and programmers.

Part II

Application Design Tuning for Designers and Programmers

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

3

Application and System Performance Characteristics

This chapter describes various applications and systems that use Oracle databases and the suggested approaches and features available when designing each type. Topics in this chapter include:

- Types of Applications
- Oracle Configurations

Types of Applications

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

- Online Transaction Processing (OLTP)
- Data Warehousing
- Multipurpose Applications

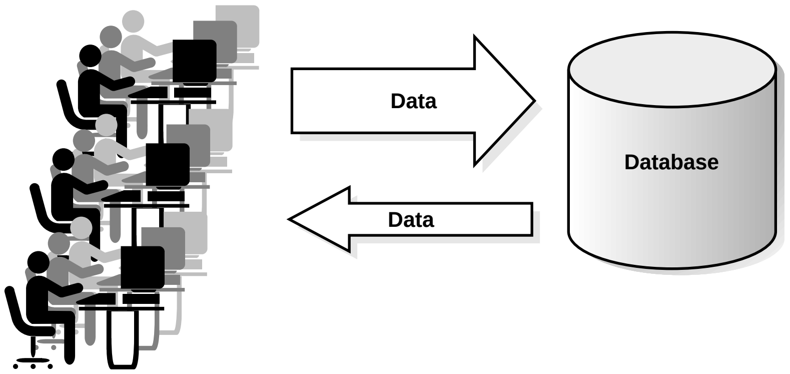
See Also: *Oracle8i Concepts*, *Oracle8i Application Developer's Guide - Fundamentals*, and *Oracle8i Administrator's Guide* for more information on these topics and how to implement their features.

Online Transaction Processing (OLTP)

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

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

Figure 3–1 Online Transaction Processing Systems



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

The following elements are crucial for tuning OLTP systems:

- Rollback segments
- Indexes, clusters, and hashing
- Discrete transactions
- Data block size
- Dynamic allocation of space to tables and rollback segments
- Transaction processing monitors and the multi-threaded server
- The shared pool
- Well-tuned SQL statements
- Integrity constraints

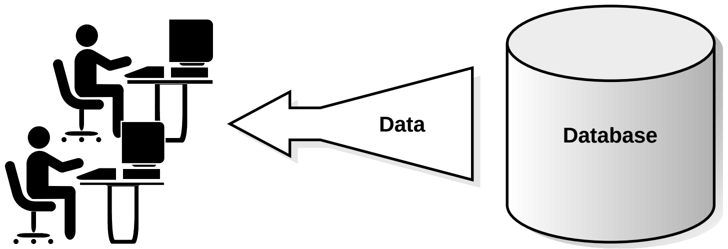
- Client/server architecture
- Dynamically changeable initialization parameters
- Procedures, packages, and functions

See Also: *Oracle8i Concepts* and *Oracle8i Administrator's Guide* for a description of each of these topics. Read about these topics before designing your system and decide which features can benefit your particular situation.

Data Warehousing

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

Figure 3–2 Data Warehousing Systems



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

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

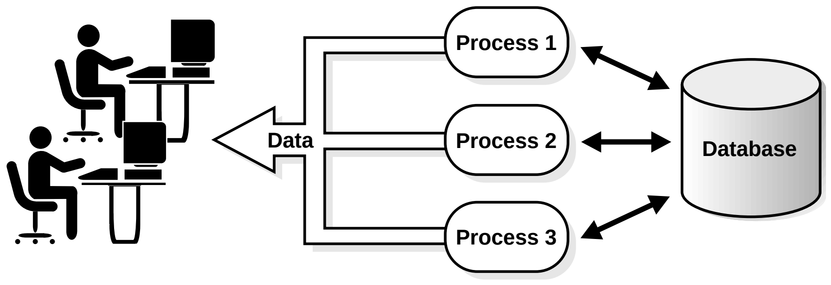
The key to performance in a decision support system is properly tuned queries and proper use of indexes, clusters, and hashing. The following issues are crucial in implementing and tuning a decision support system:

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

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

Figure 3–3 illustrates parallel execution.

Figure 3–3 Parallel Execution Processing



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

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

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

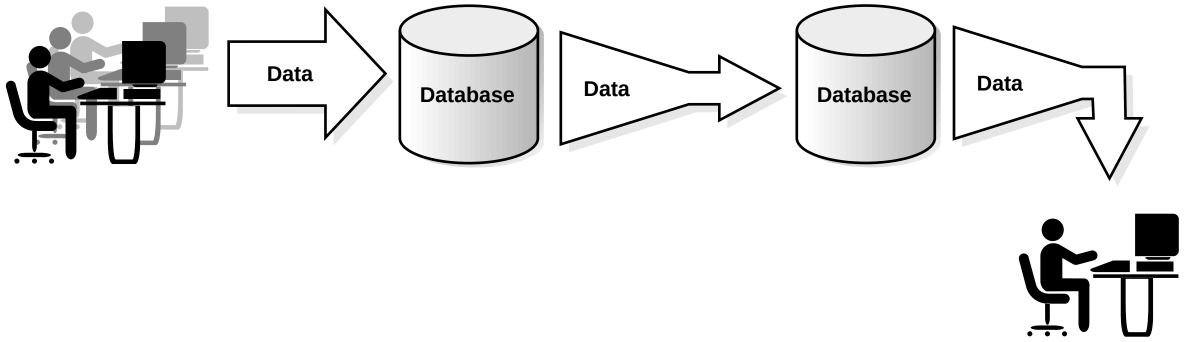
> **See Also:** Section VI, "Materialized Views", Chapter 11, "Optimizing Data Warehouse Applications" for an introduction to Oracle data warehousing functionality, Chapter 26, "Tuning Parallel Execution", for information on the performance aspects of parallel execution, and *Oracle8i Concepts* for general information about parallel execution.

Multipurpose Applications

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

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

Figure 3–4 A Hybrid OLTP/Data Warehousing System



One example of a combination OLTP / data warehousing system is a marketing tool that determines the buying patterns of consumers based on information gathered from retail stores. The retail stores gather data from daily purchase records and the marketing staff queries this data to determine which items sell best in which locations. This report is then used to determine inventory levels for particular items in each store.

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

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

Oracle Configurations

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

- Distributed Systems
- The Oracle Parallel Server
- Client / Server Configurations

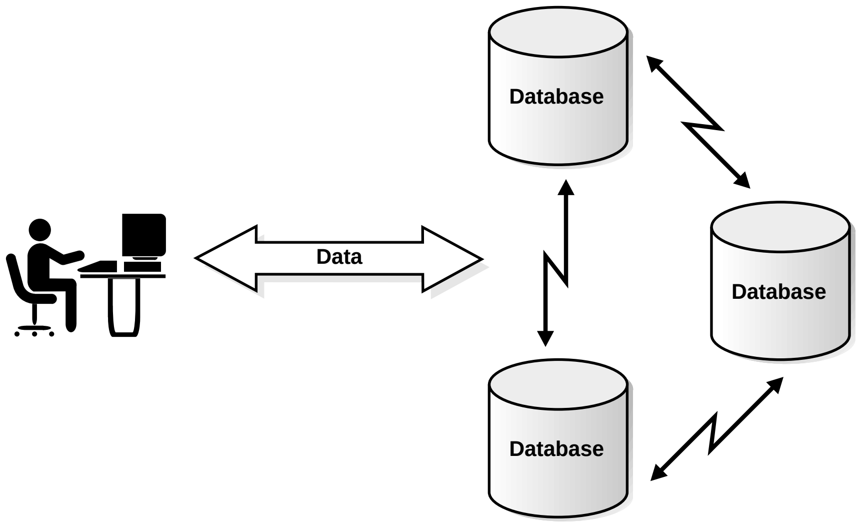
Depending on your application and your operating system, each of these or a combination of these configurations will best suit your needs.

Distributed Systems

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

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

Figure 3–5 Distributed Database System



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

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

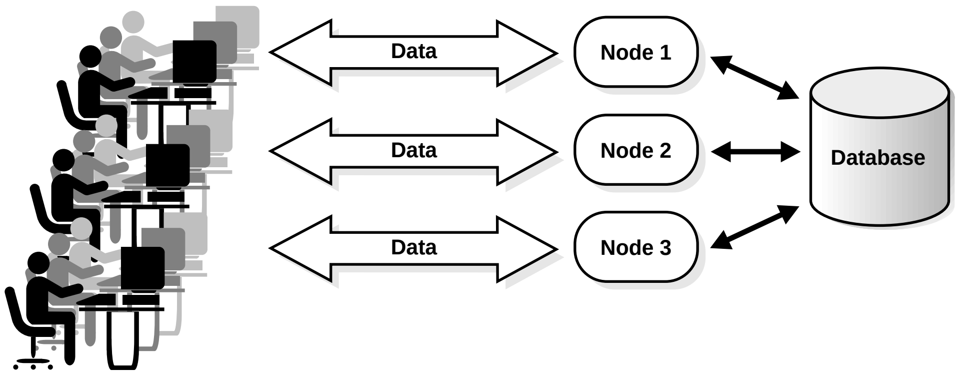
- Network configuration
- Distributed database design
- Symmetric replication
- Table snapshots and snapshot logs
- Procedures, packages, and functions

See Also: *Oracle8i Distributed Database Systems* and *Oracle8i Replication*, and Chapter 8, "Tuning Distributed Queries".

The Oracle Parallel Server

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

Figure 3–6 An Oracle Parallel Server



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

If multiple nodes require access to the same data for DML operations, that data must first be written to disk before the next node can obtain a lock. This type of contention significantly degrades performance. On such systems, data must be effectively partitioned among the various nodes for optimal performance. Read-only data can be efficiently shared across all instances in an OPS configuration without the problem of lock contention because Oracle uses a non-locking query logic.

> **See Also:** *Oracle8i Parallel Server Concepts and Administration*.

Client/Server Configurations

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

> **See Also:** "Solving CPU Problems by Changing System Architectures" on page 18-10 for information about multi-tier systems.

4
Tuning Database Operations

Structured Query Language (SQL) is used to perform all database operations, although some Oracle tools and applications simplify or mask its use. This chapter provides an overview of the issues involved in tuning database operations from the SQL point-of-view:

- Tuning Goals
- Methodology for Tuning Database Operations
- Approaches to SQL Statement Tuning

See Also: For more information about tuning PL/SQL statements, please refer to the *PL/SQL User's Guide and Reference*.

Tuning Goals

This section introduces:

- Tuning a Serial SQL Statement
- Tuning Parallel Execution
- Tuning OLTP Applications

Always approach the tuning of database operations from the standpoint of the particular goals of your application. Are you tuning serial SQL statements or parallel operations? Do you have an online transaction processing (OLTP) application, or a data warehousing application?

- Data warehousing operations process high volumes of data and they have a high correlation with the goals of parallel operations
- OLTP applications have a high transaction volume and they correlate more with serial operations

As a result, these two divergent types of applications have contrasting goals for tuning as described in Table 4–1.

Table 4–1 Contrasting Goals for Tuning

Tuning Situation	Goal
Serial SQL Statement	Minimize resource use by the operation.
Parallel Operations	Maximize throughput for the hardware.

Tuning a Serial SQL Statement

The goal of tuning one SQL statement in isolation can be stated as:

Minimize resource use by the operation being performed.

You can experiment with alternative SQL syntax without actually modifying your application. To do this, use the EXPLAIN PLAN command with the alternative statement that you are considering and compare the alternative statement’s execution plan and cost with that of the existing one. The cost of a SQL statement appears in the POSITION column of the first row generated by EXPLAIN PLAN. However, you must run the application to see which statement can actually be executed more quickly.

See Also: "Approaches to SQL Statement Tuning" on page 4-6.

Tuning Parallel Execution

The goal of tuning parallel execution can be stated as:

Maximize throughput for the given hardware.

If you have a powerful system and a massive, high-priority SQL statement to run, parallelize the statement so it use all available resources.

Oracle can perform the following operations in parallel:

- Parallel query
- Parallel DML (includes INSERT, UPDATE, DELETE; APPEND hint; parallel index scans)
- Parallel DDL
- Parallel recovery
- Parallel loading
- Parallel propagation (for replication)

Look for opportunities to parallelize operations in the following situations:

- Long elapsed time

  Whenever an operation you are performing in the database takes a long elapsed time, whether it is a query or a batch job, you may be able to reduce the elapsed time by using parallel operations.

- High number of rows processed

  You can split rows so they are not all accessed by a single process.

> **See Also:** Chapter 26, "Tuning Parallel Execution" and *Oracle8i Concepts*, for basic principles of parallel execution.

Tuning OLTP Applications

Tuning OLTP applications mostly involves tuning serial SQL statements. You should consider two design issues: use of SQL and shared PL/SQL, and use of different transaction modes.

> **See Also:** Information on tuning data warehouse applications appears in Chapter 11, "Optimizing Data Warehouse Applications".

SQL and Shared PL/SQL

To minimize parsing, use bind variables in SQL statements within OLTP applications. In this way all users can share the same SQL statements while using fewer resources for parsing.

Transaction Modes

Sophisticated users can use discrete transactions if performance is of the utmost importance, and if the users are willing to design the application accordingly.

Serializable transactions can be used if the application must be ANSI compatible. Because of the overhead inherent in serializable transactions, Oracle strongly recommends the use of read-committed transactions instead.

See Also: Chapter 9, "Transaction Modes".

Triggers

If excessive use of triggers degrades system performance, modify the conditions under which triggers fire by executing the CREATE TRIGGER or REPLACE TRIGGER commands. You can also turn off triggers with the ALTER TRIGGER command.

Note: Excessive use of triggers for frequent events such as logons, logoffs, and error events can degrade performance since these events affect all users.

Methodology for Tuning Database Operations

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

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

Step 1: Find the Statements that Consume the Most Resources

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

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

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

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

> **See Also:** Chapter 14, "The SQL Trace Facility and TKPROF", Chapter 15, "Using Oracle Trace", and *Oracle8i Reference* for more information about dynamic performance views.

Step 2: Tune These Statements To Use Fewer Resources

Remember that application design is fundamental to performance. No amount of SQL statement tuning can make up for inefficient application design. If you encounter SQL statement tuning problems, perhaps you need to change the application design.

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

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

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

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

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

Approaches to SQL Statement Tuning

This section describes five ways you can improve SQL statement efficiency:

- Restructure the Indexes
- Restructure the Statement
- Modify or Disable Triggers
- Restructure the Data
- Keeping Statistics Current and Using Plan Stability to Preserve Execution Plans

> **Note:** The guidelines described in this section are oriented to production SQL that will be executed frequently. Most of the techniques that are discouraged here can legitimately be employed in ad hoc statements or in applications run infrequently where performance is not critical.

Restructure the Indexes

Restructuring the indexes is a good starting point, because it has more impact on the application than does restructuring the statement or the data.

- Remove nonselective indexes to speed the DML
- Index performance-critical access paths
- Consider hash clusters, but watch uniqueness
- Consider index clusters only if the cluster keys are similarly sized

Do not use indexes as a panacea. Application developers sometimes think that performance will improve if they just write enough indexes. If a single programmer creates an appropriate index, this might indeed improve the application's performance. However, if 50 programmers each create an index, application performance will probably be hampered!

Restructure the Statement

After restructuring the indexes, you can try restructuring the statement. Rewriting an inefficient SQL statement is often easier than repairing it. If you understand the purpose of a given statement, you may be able to quickly and easily write a new statement that meets the requirement.

Consider Alternative SQL Syntax

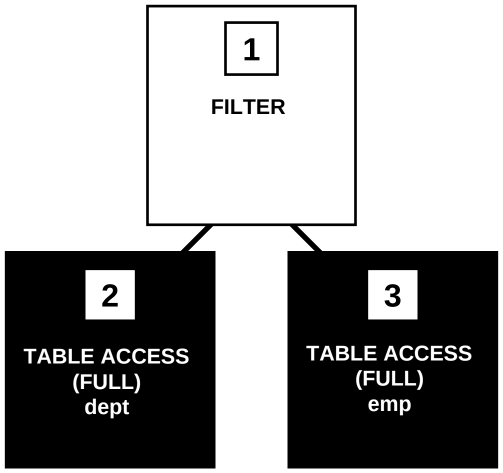
Because SQL is a flexible language, more than one SQL statement may meet the needs of your application. Although two SQL statements may produce the same result, Oracle may process one faster than the other. You can use the results of the EXPLAIN PLAN statement to compare the execution plans and costs of the two statements and determine which is more efficient.

This example shows the execution plans for two SQL statements that perform the same function. Both statements return all the departments in the DEPT table that have no employees in the EMP table. Each statement searches the EMP table with a subquery. Assume there is an index, DEPTNO_INDEX, on the DEPTNO column of the EMP table.

This is the first statement and its execution plan:

```
SELECT dname, deptno
   FROM dept
   WHERE deptno NOT IN
      (SELECT deptno FROM emp);
```

Figure 4–1 Execution Plan with Two Full Table Scans

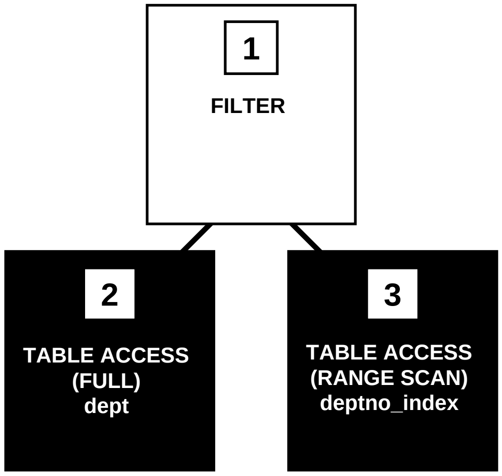


Step 3 of the output indicates that Oracle executes this statement by performing a full table scan of the EMP table despite the index on the DEPTNO column. This full table scan can be a time-consuming operation. Oracle does not use the index because the subquery that searches the EMP table does not have a WHERE clause that makes the index available.

However, this SQL statement selects the same rows by accessing the index:

```
SELECT dname, deptno
    FROM dept
   WHERE NOT EXISTS
      (SELECT deptno
         FROM emp
WHERE dept.deptno = emp.deptno);
```

Figure 4–2 Execution Plan with a Full Table Scan and an Index Scan



> **See Also:** The optimizer chapter in *Oracle8i Concepts* for more information on interpreting execution plans.

The WHERE clause of the subquery refers to the DEPTNO column of the EMP table, so the index DEPTNO_INDEX is used. The use of the index is reflected in Step 3 of the execution plan. The index range scan of DEPTNO_INDEX takes less time than the full scan of the EMP table in the first statement. Furthermore, the first query performs one full scan of the EMP table for every DEPTNO in the DEPT table. For these reasons, the second SQL statement is faster than the first.

If you have statements in your applications that use the NOT IN operator, as the first query in this example does, you should consider rewriting them so that they use the NOT EXISTS operator. This would allow such statements to use an index, if one exists.

Compose Predicates Using AND and =

Use equijoins. Without exception, statements that perform equijoins on untransformed column values are the easiest to tune.

Choose an Advantageous Join Order

Join order can have a significant effect on performance. The main objective of SQL tuning is to avoid performing unnecessary work to access rows that do not affect the result. This leads to three general rules:

- Avoid a full-table scan if it is more efficient to get the required rows through an index.
- Avoid using an index that fetches 10,000 rows from the driving table if you could instead use another index that fetches 100 rows.
- Choose the join order so as to join fewer rows to tables later in the join order.

The following example shows how to tune join order effectively:

```
SELECT info
  FROM taba a, tabb b, tabc c
 WHERE a.acol between :alow and :ahigh
   AND b.bcol between :blow and :bhigh
   AND c.ccol between :clow and :chigh
   AND a.key1 = b.key1
   AMD a.key2 = c.key2;
```

1. Choose the driving table and the driving index (if any).

   The first three conditions in the example above are filter conditions applying to only a single table each. The last two conditions are join conditions.

   Filter conditions dominate the choice of driving table and index. In general, the driving table should be the one containing the filter condition that eliminates the highest percentage of the table. Thus, if the range of :alow to :ahigh is narrow compared with the range of acol, but the ranges of :b* and :c* are relatively large, then taba should be the driving table, all else being equal.

2. Choose the right indexes.

   Once you know your driving table, choose the most selective index available to drive into that table. Alternatively, choose a full table scan if that would be more efficient. From there, the joins should all happen through the join indexes, the indexes on the primary or foreign keys used to connect that table to an earlier table in the join tree. Rarely should you use the indexes on the non-join conditions, except for the driving table. Thus, once taba is chosen as the driving table, you should use the indexes on b.key1 and c.key2 to drive into tabb and tabc, respectively.

3. Choose the best join order, driving to the best unused filters earliest.

   The work of the following join can be reduced by first joining to the table with the best still-unused filter. Thus, if "bcol between ..." is more restrictive (rejects a higher percentage of the rows seen) than "ccol between ...", the last join can be made easier (with fewer rows) if tabb is joined before tabc.

Use Untransformed Column Values

Use untransformed column values. For example, use:

```
WHERE a.order_no = b.order_no
```

Rather than:

```
WHERE TO_NUMBER (substr(a.order_no, instr(b.order_no, '.') - 1)
= TO_NUMBER (substr(a.order_no, instr(b.order_no, '.') - 1)
```

Do not use SQL functions in predicate clauses or WHERE clauses. The use of an aggregate function, especially in a subquery, often indicates that you could have held a derived value on a master record.

Avoid Mixed-type Expressions

Avoid mixed-mode expressions, and beware of implicit type conversions. When you want to use an index on the VARCHAR2 column *charcol*, but the WHERE clause looks like this:

```
AND charcol = <numexpr>
```

Where *numexpr* is an expression of number type (for example, 1, USERENV('SESSIONID'), numcol, numcol+0,...), Oracle will translate that expression into:

```
AND to_number(charcol) = numexpr
```

This has the following consequences:

- Any expression using a column, such as a function having the column as its argument, will cause the optimizer to ignore the possibility of using an index on that column, even a unique index.
- If the system processes even a single row having charcol as a string of characters that does not translate to a number, an error will be returned.

You can avoid this problem by replacing the top expression with the explicit conversion:

```
AND charcol = to_char(<numexpr>)
```

Alternatively, make all type conversions explicit. The statement:

```
numcol = charexpr
```

allows use of an index on *numcol* because the default conversion is always character-to-number. This behavior, however, is subject to change. Making type conversions explicit also makes it clear that *charexpr* should always translate to a number.

Write Separate SQL Statements for Specific Values

SQL is not a procedural language. Using one piece of SQL to do many different things is not a good idea: it usually results in a less-than-optimal result for each task. If you want SQL to accomplish different things, then write two different statements rather than writing one statement that will do different things depending on the parameters you give it.

Optimization (determining the execution plan) takes place before the database knows what values will be substituted into the query. An execution plan should not, therefore, depend on what those values are. For example:

```
SELECT info from tables
 WHERE ...
   AND somecolumn BETWEEN decode(:loval, 'ALL', somecolumn, :loval)
   AND decode(:hival, 'ALL', somecolumn, :hival);
```

Written as shown, the database cannot use an index on the *somecolumn* column because the expression involving that column uses the same column on both sides of the BETWEEN.

This is not a problem if there is some other highly selective, indexable condition you can use to access the driving table. Often, however, this is not the case. Frequently you may want to use an index on a condition like that shown, but need to know the values of :loval, and so on, in advance. With this information you can rule out the ALL case, which should *not* use the index.

If you want to use the index whenever real values are given for :loval and :hival (that is, if you expect narrow ranges, even ranges where :loval often equals :hival), 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, you seem to obtain both a desirable and an undesirable execution plan. However, the first condition the database evaluates for either half of the UNION ALL will be 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. Since the final conditions on :hival and :loval are guaranteed to be mutually exclusive, then only one half of the UNION ALL will actually return rows. (The ALL in UNION ALL is logically valid because of this exclusivity. It allows the plan to be carried out without an expensive sort to rule out duplicate rows for the two halves of the query.)

Use Hints to Control Access Paths

Use optimizer hints, such as `/*+ORDERED */` to control access paths. This is a better approach than using traditional techniques or "tricks of the trade" such as CUST_NO + 0. For example, use

```
SELECT /*+ FULL(EMP) */ E.ENAME
  FROM EMP E
WHERE E.JOB = 'CLERK';
```

rather than

```
SELECT E.ENAME
  FROM EMP E
WHERE E.JOB || '' = 'CLERK';
```

See Also: For more information on hints, please refer to Chapter 7, "Optimizer Modes, Plan Stability, and Hints".

Use Care When Using IN and NOT IN with a Subquery

Remember that WHERE (NOT) EXISTS is a useful alternative.

Use Care When Embedding Data Value Lists in Applications

Data value lists are normally a sign that an entity is missing. For example:

```
WHERE TRANSPORT IN ('BMW', 'CITROEN', 'FORD', HONDA')
```

The real objective in the WHERE clause above is to determine whether the mode of transport is an automobile, and not to identify a particular make. A reference table should be available in which transport type='AUTOMOBILE'.

Minimize the use of DISTINCT. DISTINCT always creates a SORT; all the data must be instantiated before your results can be returned.

Reduce the Number of Calls to the Database

When appropriate, use INSERT, UPDATE, or DELETE . . . RETURNING to select and modify data with a single call. This technique improves performance by reducing the number of calls to the database.

See Also: *Oracle8i SQL Reference* for syntax information on the INSERT, UPDATE, and DELETE commands.

Use Care When Managing Views

Be careful when joining views, when performing outer joins to views, and when you consider recycling views.

Use Care When Joining Views The shared SQL area in Oracle reduces the cost of parsing queries that reference views. In addition, optimizer improvements make the processing of predicates against views very efficient. Together these factors make possible the use of views for ad hoc queries. Despite this, joins to views are not recommended, particularly joins from one complex view to another.

The following example shows a query upon a column which is the result of a GROUP BY. The entire view is first instantiated, and then the query is run against the view data.

```
CREATE VIEW DX(deptno, dname, totsal)
  AS SELECT D.deptno, D.dname, E.sum(sal)
       FROM emp E, dept D
      WHERE E.deptno = D.deptno
    GROUP BY deptno, dname
SELECT * FROM DX WHERE deptno=10;
```

Use Care When Performing Outer Joins To Views An outer join to a multitable view can be problematic. For example, you may start with the usual emp and dept tables with indexes on e.empno, e.deptno, and d.deptno, and create the following view:

```
CREATE VIEW EMPDEPT (EMPNO, DEPTNO, ename, dname)
  AS SELECT E.EMPNO, E.DEPTNO, e.ename, d.dname
       FROM DEPT D, EMP E
       WHERE E.DEPTNO = D.DEPTNO(+);
```

You may then construct the simplest possible query to do an outer join into this view on an indexed column (e.deptno) of a table underlying the view:

```
SELECT e.ename, d.loc
  FROM dept d, empdept e
 WHERE d.deptno = e.deptno(+)
    AND d.deptno = 20;
```

The following execution plan results:

```
QUERY_PLAN
--------------------------------------------
MERGE JOIN OUTER
 TABLE ACCESS BY ROWID DEPT
  INDEX UNIQUE SCAN DEPT_U1: DEPTNO
 FILTER
  VIEW EMPDEPT
   NESTED LOOPS OUTER
    TABLE ACCESS FULL EMP
    TABLE ACCESS BY ROWID DEPT
      INDEX UNIQUE SCAN DEPT_U1: DEPTNO
```

Until both tables of the view are joined, the optimizer does not know whether the view will generate a matching row. The optimizer must therefore generate *all* the rows of the view and perform a MERGE JOIN OUTER with all the rows returned from the rest of the query. This approach would be extremely inefficient if all you want is a few rows from a multitable view with at least one very large table.

To solve this problem is relatively easy, in the preceding example. The second reference to dept is not needed, so you can do an outer join straight to emp. In other cases, the join need not be an outer join. You can still use the view simply by getting rid of the (+) on the join into the view.

Do Not Recycle Views Beware of writing a view for one purpose and then using it for other purposes, to which it may be ill-suited. Consider this example:

```
SELECT dname from DX
7 WHERE deptno=10;
```

You can obtain dname and deptno directly from the DEPT table. It would be inefficient to obtain this information by querying the DX view (which was declared earlier in the present example). To answer the query, the view would perform a join of the DEPT and EMP tables, even though you do not need any data from the EMP table.

Modify or Disable Triggers

Using triggers consumes system resources. If you use too many triggers, you may find that performance is adversely affected and you may need to modify or disable them.

Restructure the Data

After restructuring the indexes and the statement, you can consider restructuring the data.

- Introduce derived values. Avoid GROUP BY in response-critical code
- Implement missing entities and intersection tables
- Reduce the network load. Migrate, replicate, partition data

The overall purpose of any strategy for data distribution is to locate each data attribute such that its value makes the minimum number of network journeys. If the current number of journeys is excessive, then moving (migrating) the data is a natural solution.

Often, however, no single location of the data reduces the network load (or message transmission delays) to an acceptable level. In this case, consider either holding multiple copies (replicating the data) or holding different parts of the data in different places (partitioning the data).

Where distributed queries are necessary, it may be effective to code the required joins with procedures either in PL/SQL within a stored procedure, or within the user interface code.

When considering a cross-network join, you can either bring the data in from a remote node and perform the join locally, or you can perform the join remotely. The option you choose should be determined by the relative volume of data on the different nodes.

Keeping Statistics Current and Using Plan Stability to Preserve Execution Plans

Once you have tuned your application's SQL statements, consider maintaining statistics with the useful procedures of the DBMS_STATS package. Also consider implementing Plan Stability features to maintain application performance characteristics despite system changes. Both of these topics are discussed in Chapter 7, "Optimizer Modes, Plan Stability, and Hints".

5

Registering Applications

Application developers can use the DBMS_APPLICATION_INFO package with Oracle Trace and the SQL trace facility to record names of executing modules or transactions in the database for later use when tracking the performance of various modules. This chapter describes how to register an application with the database and retrieve statistics on each registered module or code segment.

Oracle provides a method for applications to register the name of the application and actions performed by that application with the database. Registering the application allows system administrators and performance tuning specialists to track performance by module. System administrators can also use this information to track resource use by module. When an application registers with the database, its name and actions are recorded in the V$SESSION and V$SQLAREA views.

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

Topics in this chapter include:

DBMS_APPLICATION_INFO Package

To register applications with the database, use the procedures in the DBMS_APPLICATION_INFO package. DBMS_APPLICATION_INFO provides the following procedures:

Table 5–1 Procedures in the DBMS_APPLICATION_INFO Package

Procedure	Description
SET_MODULE	Sets the name of the module that is currently running.
SET_ACTION	Sets the name of the current action within the current module.
SET_CLIENT_INFO	Sets the client information field for the session.
READ_MODULE	Reads values of module and action fields for the current session.
READ_CLIENT_INFO	Reads the client information field for the current session.

Privileges

Before using this package, you must run the DBMSUTL.SQL script to create the DBMS_APPLICATION_INFO package.

> **See Also:** For more information about Oracle supplied packages and executing stored procedures, see the *Oracle8i Supplied Packages Reference.*

Setting the Module Name

To set the name of the current application or module, use the SET_MODULE procedure in the DBMS_APPLICATION_INFO package. The module name should be the name of the procedure (if using stored procedures), or the name of the application. The action name should describe the action performed.

Example

The sample PL/SQL block in the following SQL statement, starting at the BEGIN keyword, sets the module name and action name:

```
CREATE PROCEDURE add_employee(
   name       VARCHAR2(20),
   salary     NUMBER(7,2),
   manager    NUMBER,
   title      VARCHAR2(9),
   commission NUMBER(7,2),
   department NUMBER(2))  AS
BEGIN
  DBMS_APPLICATION_INFO.SET_MODULE(
      module_name => 'add_employee',
      action_name => 'insert into emp');
  INSERT INTO emp
     (ename, empno, sal, mgr, job, hiredate, comm, deptno)
     VALUES (name, next.emp_seq, manager, title, SYSDATE,
             commission, department);
  DBMS_APPLICATION_INFO.SET_MODULE('','');
END;
```

Syntax

Syntax and parameters for the SET_MODULE procedure are described here:

```
DBMS_APPLICATION_INFO.SET_MODULE(
   module_name   IN VARCHAR2,
   action_name   IN VARCHAR2)
```

module_name	Name of module that is currently running. When the current module terminates, call this procedure with the name of the new module if there is one, or null if there is not. Names longer than 48 bytes are truncated.
action_name	Name of current action within the current module. If you do not want to specify an action, this value should be null. Names longer than 32 bytes are truncated.

Setting the Action Name

To set the name of the current action within the current module, use the SET_ACTION command in the DBMS_APPLICATION_INFO package. The action name should be descriptive text about the current action being performed. You should probably set the action name before the start of every transaction.

Example

The following is an example of a transaction that uses the registration procedure:

```
 CREATE OR REPLACE PROCEDURE bal_tran (amt IN NUMBER(7,2)) AS
 BEGIN
-- balance transfer transaction
```

```
    DBMS_APPLICATION_INFO.SET_ACTION(
      action_name => 'transfer from chk to sav');
    UPDATE chk SET bal = bal + :amt
      WHERE acct# = :acct;
    UPDATE sav SET bal = bal - :amt
      WHERE acct# = :acct;
    COMMIT;
    DBMS_APPLICATION_INFO.SET_ACTION('');
END;
```

Set the transaction name to null after the transaction completes so that subsequent transactions are logged correctly. If you do not set the transaction name to null, subsequent transactions may be logged with the previous transaction's name.

Syntax

The parameter for the SET_ACTION procedure is described in this section. The syntax for this procedure is shown below:

```
DBMS_APPLICATION_INFO.SET_ACTION(action_name IN VARCHAR2)
```

action_name	The name of the current action within the current module. When the current action terminates, call this procedure with the name of the next action if there is one, or null if there is not. Names longer than 32 bytes are truncated.

Setting the Client Information

To supply additional information about the client application, use the SET_CLIENT_INFO procedure in the DBMS_APPLICATION_INFO package.

Syntax

The parameter for the SET_CLIENT_INFO procedure is described in this section. The syntax for this procedure is shown below:

```
DBMS_APPLICATION_INFO.SET_CLIENT_INFO(client_info IN VARCHAR2)
```

client_info	Use this parameter to supply any additional information about the client application. This information is stored in the V$SESSIONS view. Information exceeding 64 bytes is truncated.

Retrieving Application Information

Module and action names for a registered application can be retrieved by querying V$SQLAREA or by calling the READ_MODULE procedure in the DBMS_APPLICATION_INFO package. Client information can be retrieved by querying the V$SESSION view, or by calling the READ_CLIENT_INFO procedure in the DBMS_APPLICATION_INFO package.

Querying V$SQLAREA

The following sample query illustrates the use of the MODULE and ACTION column of the V$SQLAREA.

```
   SELECT sql_text, disk_reads, module, action
     FROM v$sqlarea
     WHERE module = ’add_employee’;

SQL_TEXT                   DISK_READS  MODULE             ACTION
-------------------        ----------  ------------------ ----------------
INSERT INTO emp            1           add_employee       insert into emp
(ename, empno, sal,
mgr, job, hiredate,
comm, deptno)
VALUES
(name,
next.emp_seq,
manager, title,
SYSDATE, commission,
department)

1 row selected.
```

READ_MODULE Syntax

The parameters for the READ_MODULE procedure are described in this section. The syntax for this procedure is shown below:

```
 DBMS_APPLICATION_INFO.READ_MODULE(
    module_name   OUT   VARCHAR2,
    action_name   OUT   VARCHAR2)
```

module_name	The last value that the module name was set to by calling SET_MODULE.
action_name	The last value that the action name was set to by calling SET_ACTION or SET_MODULE.

READ_CLIENT_INFO Syntax

The parameter for the READ_CLIENT_INFO procedure is described in this section. The syntax for this procedure is shown below:

```
DBMS_APPLICATION_INFO.READ_CLIENT_INFO(client_info OUT VARCHAR2)
```

client_info	The last client information value supplied to the SET_CLIENT_INFO procedure.

6

Data Access Methods

This chapter provides an overview of data access methods that can enhance performance. It also warns of situations to avoid. This chapter also explains how to used hints to force various approaches. Topics in this chapter include:

Using Indexes

This section describes:

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

When to Create Indexes

Indexes improve the performance of queries that select a small percentage of rows from a table. As a general guideline, create indexes on tables that are queried for less than 2% or 4% of the table's rows. This value may be higher in situations where all data can be retrieved from an index, or where the indexed columns and expressions can be used for joining to other tables.

This guideline is based on these assumptions:

- Rows with the same value for the key on which the query is based are uniformly distributed throughout the data blocks allocated to the table
- Rows in the table are randomly ordered with respect to the key on which the query is based
- The table contains a relatively small number of columns
- Most queries on the table have relatively simple WHERE clauses
- The cache hit ratio is low and there is no operating system cache

If these assumptions do not describe the data in your table and the queries that access it, then an index may not be helpful unless your queries typically access at least 25% of the table's rows.

Tuning the Logical Structure

Although cost-based optimization helps avoid the use of nonselective indexes within query execution, the SQL engine must continue to maintain all indexes defined against a table regardless of whether they are used. Index maintenance can present a significant CPU and I/O resource demand in any I/O intensive application. Put another way, building indexes "just in case" is not a good practice; indexes should not be built until required.

To maintain optimal performance as far as indexes are concerned, drop indexes that your application is not using. You can find indexes that are not referenced in execution plans by processing all of your application SQL through EXPLAIN PLAN and capturing the resulting plans. Unused indexes are typically, though not necessarily, nonselective.

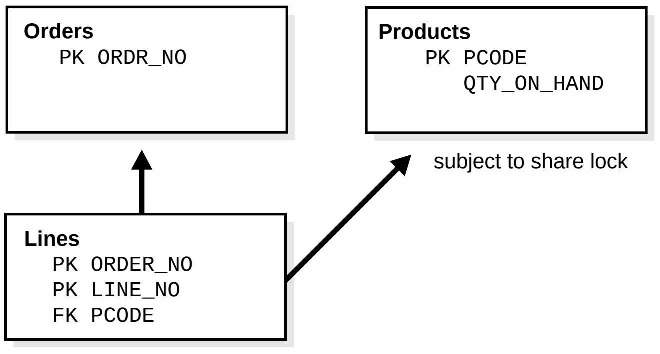
Indexes within an application sometimes have uses that are not immediately apparent from a survey of statement execution plans. In particular, Oracle uses "pins" (nontransactional locks) on foreign key indexes to avoid using shared locks on the child table when enforcing foreign key constraints.

In many applications a foreign key index never, or rarely, supports a query. In the example shown in Figure 6–1, the need to locate all of the order lines for a given product may never arise. However, when no index exists with LINES(PCODE) as its leading portion (as described in "Choosing Composite Indexes"), then Oracle places a share lock on the LINES table each time PRODUCTS(PCODE) is updated or deleted. Such a share lock is a problem only if the PRODUCTS table is subject to frequent DML.

If this contention arises, then to remove it the application must either:

- Accept the additional load of maintaining the index
- Accept the risk of running with the constraint disabled

Figure 6–1 Foreign Key Constraint



Choosing Columns and Expressions to Index

A key is a column or expression on which you can build an index. Follow these guidelines for choosing index keys to index:

- Consider indexing keys that are frequently used in WHERE clauses.
- Consider indexing keys that are frequently used to join tables in SQL statements. For more information on optimizing joins, see the section "Using Hash Clusters" on page 6-25.
- Only index keys that have accurate 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 on the keys and expressions of unique and primary keys that you define with integrity constraints. These indexes are the most selective and the most effective in optimizing performance.

  You can determine the selectivity of an index by dividing the number of rows in the table by the number of distinct indexed values. You can obtain these values using the ANALYZE statement. Selectivity calculated in this manner should be interpreted as a percentage.

- Do not use standard B*-tree indexes on keys or expressions with few distinct values. Such keys or expressions usually have poor selectivity and therefore do not optimize performance unless the frequently selected key values appear less frequently than the other key values. You can use bitmap indexes effectively in such cases, unless a high concurrency OLTP application is involved.

- Do not index columns that are frequently modified. UPDATE statements that modify indexed columns and INSERT and DELETE statements that modify indexed tables take longer than if there were no index. Such SQL statements must modify data in indexes as well as data in tables. They also generate additional undo and redo information.
- Do not index keys that appear only in WHERE clauses with functions or operators. A WHERE clause that uses a function (other than MIN or MAX) or an operator with an indexed key does not make available the access path that uses the index.
- Consider indexing foreign keys of referential integrity constraints in cases in which a large number of concurrent INSERT, UPDATE, and DELETE statements access the parent and child tables. Such an index allows UPDATEs and DELETEs on the parent table without share locking the child table.
- When choosing to index a key, consider whether the performance gain for queries is worth the performance loss for INSERTs, UPDATEs, and DELETEs and the use of the space required to store the index. You may want to experiment by comparing the processing times of your SQL statements with and without indexes. You can measure processing time with the SQL trace facility.

See Also: *Oracle8i Concepts* regarding 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 more accurate selectivity.
Additional data storage	If all columns selected by a query are in a composite index, Oracle can return these values from the index without accessing the table.

A SQL statement can use an access path involving a composite index if the statement contains constructs that use a leading portion of the index. A leading portion of an index is a set of one or more columns that were specified first and consecutively in the list of columns in the CREATE INDEX statement that created the index. Consider this CREATE INDEX statement:

```
CREATE INDEX comp_ind
    ON tab1(x, y, z);
```

These combinations of columns are leading portions of the index: X, XY, and XYZ. These combinations of columns are not leading portions of the index: YZ, Z, and Y.

Follow these guidelines for choosing keys for composite indexes:

- Consider creating a composite index on keys that are frequently used together in WHERE clause conditions combined with AND operators, especially if their combined selectivity is better than the selectivity of either key individually.
- If several queries select the same set of keys based on one or more key values, consider creating a composite index containing all of these keys.

Of course, consider the guidelines associated with the general performance advantages and trade-offs of indexes described in the previous sections. Follow these guidelines for ordering keys in composite indexes:

- Create the index so the keys used in WHERE clauses make up a leading portion.
- If some keys are used in WHERE clauses more frequently, be sure to create the index so that the more frequently selected keys make up a leading portion to allow the statements that use only these keys to use the index.
- If all keys are used in WHERE clauses equally often, ordering these keys from most selective to least selective in the CREATE INDEX statement best improves query performance.
- If all keys are used in the WHERE clauses equally often but the data is physically ordered on one of the keys, place that key first in the composite index.

Writing Statements that Use Indexes

Even after you create an index, the optimizer cannot use an access path that uses the index simply because the index exists. The optimizer can choose such an access path for a SQL statement only if it contains a construct that makes the access path available.

To be sure that a SQL statement can use an access path that uses an index, be sure the statement contains a construct that makes such an access path available. If you are using the cost-based approach, also generate statistics for the index. Once you have made the access path available for the statement, the optimizer may or may not choose to use the access path, based on the availability of other access paths.

If you create new indexes to tune statements, 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, Oracle invalidates the statement. When the statement is next executed, the optimizer automatically chooses a new execution plan that could potentially use the new index. If you create new indexes on a remote database to tune a distributed statement, the optimizer considers these indexes when the statement is next parsed.

Also keep in mind that the way you tune one statement may affect the optimizer's choice of execution plans for others. For example, if you create an index to be used by one statement, the optimizer may choose to use that index for other statements in your application as well. For this reason, you should re-examine your application's performance and rerun the SQL trace facility after you have tuned those statements that you initially identified for tuning.

Writing Statements that Avoid Using Indexes

In some cases, you may want to prevent a SQL statement from using an access path that uses an existing index. You may want to do this if you know that the index is not very selective and that a full table scan would be more efficient. If the statement contains a construct that makes such an index access path available, you can force the optimizer to use a full table scan through one of these methods:

- You can make the index access path unavailable by modifying the statement in a way that does not change its meaning.
- You can use the FULL hint to force the optimizer to choose a full table scan instead of an index scan.
- You can use the INDEX, INDEX_COMBINE, or AND_EQUAL hints to force the optimizer to use one index or a set of listed indexes instead of another.

The behavior of the optimizer may change in future versions of Oracle, so relying on methods such as the first to choose access paths may not be a good long-range plan. Instead, use hints to suggest specific access paths to the optimizer.

Assessing the Value of Indexes

A crude way to determine whether an index is good is to create it, analyze it, and use EXPLAIN PLAN on your query to see if the optimizer uses it. If it does, keep the index unless it is expensive to maintain. This method, however, is very time- and resource-consuming. A preferable method is to compare the optimizer cost (in the first row of EXPLAIN PLAN output) of the plans with and without the index.

Parallel execution uses indexes effectively. It does not perform parallel index range scans, but it does perform parallel index lookups for parallel nested loop join execution. If an index is very selective (there are few rows per index entry), then it may be better to use sequential index lookup than parallel table scan.

Using Fast Full Index Scans

The fast full index scan is an alternative to a full table scan when there is an index that contains all the keys that are needed for the query. A fast full scan is faster than a normal full index scan in that it can use multiblock I/O and can be parallelized just like a table scan. Unlike regular index scans, however, you cannot use keys and the rows will not necessarily come back in sorted order. The following query and plan illustrate this feature.

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

The plan is as follows:

```
SELECT STATEMENT
    SORT AGGREGATE
        HASH JOIN
            TABLE ACCESS1FULL
            INDEXT2_C1_IDX FAST FULL SCAN
```

Since index T2_C1_IDX contains all columns needed from table T2(C2), the optimizer uses a fast full index scan on that index.

Fast full scan has the following restrictions:

- At least one indexed column of the table must have the NOT NULL constraint.
- There must be a parallel clause on the index, if you want to perform fast full index scan in parallel. The parallel degree of the index is set independently: the index does *not* inherit the degree of parallelism of the table.
- Make sure you have analyzed the index, otherwise the optimizer may decide not to use it.

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

See Also: "INDEX_FFS" on page 7-48.

Re-creating Indexes

You may wish to re-create an index to compact it and minimize fragmented space, or to change the index's storage characteristics. When creating a new index that is a subset of an existing index, or when rebuilding an existing index with new storage characteristics, Oracle may use the existing index instead of the base table to improve performance.

However, there are cases where it may be beneficial to use the base table instead of the existing index. Consider an index on a table on which a lot of DML has been performed. Because of the DML, the size of the index may increase to the point where each block is only 50% full, or even maybe less. If the index refers to most of the columns in the table, the index could actually be *larger* than the table. In this case, it is faster to use the base table rather than the index to re-create the index. Another option is to create a new index on a subset of the columns of the original index.

Consider, for example, a table named CUST with columns NAME, CUSTID, PHONE, ADDR, BALANCE, and an index named I_CUST_CUSTINFO on table columns NAME, CUSTID and BALANCE. To create a new index named I_CUST_CUSTNO on columns CUSTID and NAME, you would enter:

```
CREATE INDEX I_CUST_CUSTNO ON CUST(CUSTID,NAME);
```

Oracle automatically uses the existing index (I_CUST_CUSTINFO) to create the new index rather than accessing the entire table. The syntax used is the same as if the index I_CUST_CUSTINFO did not exist.

Similarly, if you have an index on the EMPNO and MGR columns of the EMP table, and you want to change the storage characteristics of that composite index, Oracle can use the existing index to create the new index.

Use the ALTER INDEX ... REBUILD statement to reorganize or compact an existing index or to change its storage characteristics. The REBUILD statement uses the existing index as the basis for the new one. All index storage statements are supported, such as STORAGE (for extent allocation), TABLESPACE (to move the index to a new tablespace), and INITRANS (to change the initial number of entries).

ALTER INDEX ... REBUILD is usually faster than dropping and re-creating an index, because this statement uses the fast full scan feature. It reads all the index blocks using multiblock I/O then discards the branch blocks. A further advantage of this approach is that the old index is still available for queries (but not for DML) while the rebuild is in progress.

> **See Also:** *Oracle8i SQL Reference* for more information about the CREATE INDEX and ALTER INDEX statements and for restrictions on re-building indexes.

Compacting Indexes

You can coalesce leaf blocks of an index using the ALTER INDEX statement with the COALESCE option. This allows you to combine leaf levels of an index to free blocks for re-use. You can also rebuild the index online.

For more information about the syntax for this statement, please refer to the *Oracle8i SQL Reference* and the *Oracle8i Administrator's Guide.*

Using Nonunique Indexes to Enforce Uniqueness

You can use an existing nonunique index on a table to enforce uniqueness, either for UNIQUE constraints or the unique aspect of a PRIMARY KEY constraint. The advantage of this approach is that the index remains available and valid when the constraint is disabled. Therefore, enabling a disabled UNIQUE or PRIMARY KEY constraint does not require rebuilding the unique index associated with the constraint. This can yield significant time savings on enable operations for large tables.

Using a nonunique index to enforce uniqueness also allows you to 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 may use the existing index to enable and enforce the constraint. You also save significant space by not duplicating the index.

Using Enabled Novalidated Constraints

An enabled novalidated constraint behaves similarly to an enabled validated constraint. Placing a constraint in the enabled novalidated state signifies that any new data entered into the table must conform to the constraint. Existing data is not checked. Placing a constraint in the enabled novalidated state allows you to enable the constraint without locking the table.

If you change a constraint from disabled to enabled, the table must be locked. No new DML, queries, or DDL can occur because there is no mechanism to ensure that operations on the table conform to the constraint during the enable operation. The enabled novalidated state prevents operations violating the constraint from being performed on the table.

An enabled novalidated constraint can be validated with a parallel, consistent-read query of the table to determine whether any data violates the constraint. No locking is performed and the enable operation does not block readers or writers to the table. In addition, enabled novalidated constraints can be validated in parallel: multiple constraints can be validated at the same time and each constraint's validity check can be determined using parallel query.

Use the following approach to create tables with constraints and indexes:

1. Create the tables with the constraints. NOT NULL constraints may be unnamed and should be created enabled and validated. All other constraints (CHECK, UNIQUE, PRIMARY KEY, and FOREIGN KEY) should be named and should be "created disabled".

   Note: By default, constraints are created in the ENABLED state.

2. Load old data into the tables.
3. Create all indexes including indexes needed for constraints.
4. Enable novalidate all constraints. Do this to primary keys before foreign keys.
5. Allow users to query and modify data.
6. With a separate ALTER TABLE statement for each constraint, validate all constraints. Do this to primary keys before foreign keys.

For example,

```
CREATE TABLE t (a NUMBER CONSTRAINT apk PRIMARY KEY DISABLE,
   b NUMBER NOT NULL);
CREATE TABLE x (c NUMBER CONSTRAINT afk REFERENCES t DISABLE);
```

At this point, use import or fast loader to load data into t.

```
CREATE UNIQUE INDEX tai ON t (a);
CREATE INDEX tci ON x (c);
ALTER TABLE t MODIFY CONSTRAINT apk ENABLE NOVALIDATE;
ALTER TABLE x MODIFY CONSTRAINT afk ENABLE NOVALIDATE;
```

Now users can start performing inserts, updates, deletes, and selects on t.

```
ALTER TABLE t ENABLE CONSTRAINT apk;
ALTER TABLE x ENABLE CONSTRAINT afk;
```

Now the constraints are enabled and validated.

See Also: *Oracle8i Concepts* for a complete discussion of integrity constraints.

Using Function-based Indexes

A function-based index is an index on an expression. Oracle strongly recommends using function-based indexes whenever possible. Define function-based indexes anywhere where you use an index on a column, except for columns with LOBs, or REFs. Nested table columns and object types cannot contain these columns.

You can create function-based indexes for any repeatable SQL function. Oracle recommends using function-based indexes for range scans and for functions in ORDER BY clauses.

Function-based indexes are an efficient mechanism for evaluating statements that contain functions in WHERE clauses. You can create a function-based index to materialize computational-intensive expressions in the index. This permits Oracle to bypass computing the value of the expression when processing SELECT and DELETE statements. When processing INSERT and UPDATE statements, however, Oracle evaluates the function to process the statement.

For example, if you create the following index:

```
CREATE INDEX idx ON table_1 (a + b * (c - 1), a, b);
```

Oracle can use it when processing queries such as:

```
SELECT a FROM table_1 WHERE a + b * (c - 1) < 100;
```

Function-based indexes defined with the UPPER(column_name) or LOWER(column_name) keywords allow case-insensitive searches. For example, the following index:

```
CREATE INDEX uppercase_idx ON emp (UPPER(empname));
```

Facilitates processing queries such as:

```
SELECT * FROM emp WHERE UPPER(empname) = 'MARK';
```

You can also use function-based indexes for NLS sort indexes that provide efficient linguistic collation in SQL statements.

Oracle treats indexes with columns marked DESC as function-based indexes. The columns marked DESC are sorted in descending order.

Note: You must set the session parameter QUERY_REWRITE_ENABLED to TRUE to enable function-based indexes for queries. If QUERY_REWRITE_ENABLED is FALSE, function-based indexes will not be used for obtaining the values of an expression in the function-based index. However, function-based indexes can still be used for obtaining values in real columns.

Function-based Indexes and Index Organized Tables

The secondary index on an IOT can be a function-based index.

Using Bitmap Indexes

This section describes:

- When to Use Bitmap Indexes
- Creating Bitmap Indexes
- Initialization Parameters for Bitmap Indexing
- Using Bitmap Access Plans on Regular B*-tree Indexes
- Estimating Bitmap Index Size
- Bitmap Index Restrictions

See Also: *Oracle8i Concepts*, for a general introduction to bitmap indexing.

When to Use Bitmap Indexes

This section describes three aspects of indexing that you must evaluate when considering whether to use bitmap indexing on a given table: performance, storage, and maintenance.

Performance Considerations

Bitmap indexes can substantially improve performance of queries with the following characteristics:

- The WHERE clause contains multiple predicates on low- or medium-cardinality columns
- The individual predicates on these low- or medium-cardinality columns select a large number of rows
- Bitmap indexes have been created on some or all of these low- or medium-cardinality columns
- The tables being queried contain many rows

You can use multiple bitmap indexes to evaluate the conditions on a single table. Bitmap indexes are thus highly advantageous for complex ad hoc queries that contain lengthy WHERE clauses. Bitmap indexes can also provide optimal performance for aggregate queries and for optimizing joins in star schemas.

> **See Also:** For more information, please refer to the optimizing anti-joins and semi-joins discussion in *Oracle8i Concepts*.

Storage Considerations

Bitmap indexes can provide considerable storage savings over the use of multicolumn (or concatenated) B*-tree indexes. In databases containing only B*-tree indexes, you must anticipate the columns that would commonly be accessed together in a single query, and create a composite B*-tree index on these columns.

Not only would this B*-tree index require a large amount of space, but it would also be ordered. That is, a B*-tree index on (MARITAL_STATUS, REGION, GENDER) is useless for queries that only access REGION and GENDER. To completely index the database, you must create indexes on the other permutations of these columns. For the simple case of three low-cardinality columns, there are six possible composite B*-tree indexes. You must consider the trade-offs between disk space and performance needs when determining which composite B*-tree indexes to create.

Bitmap indexes solve this dilemma. Bitmap indexes can be efficiently combined during query execution, so three small single-column bitmap indexes can do the job of six three-column B*-tree indexes.

Bitmap indexes are much more efficient than B*-tree indexes, especially in data warehousing environments. Bitmap indexes are created not only for efficient space usage, but also for efficient execution, and the latter is somewhat more important.

If a bitmap index is created on a unique key column, it requires more space than a regular B*-tree index. However, for columns where each value is repeated hundreds or thousands of times, a bitmap index typically is less than 25% of the size of a regular B*-tree index. The bitmaps themselves are stored in compressed format.

Simply comparing the relative sizes of B*-tree and bitmap indexes is not an accurate measure of effectiveness, however. Because of their different performance characteristics, you should keep B*-tree indexes on high-cardinality data, while creating bitmap indexes on low-cardinality data.

Maintenance Considerations

Bitmap indexes benefit data warehousing applications but they are not appropriate for OLTP applications with a heavy load of concurrent INSERTs, UPDATEs, and DELETEs. In a data warehousing environment, data is usually maintained by way of bulk inserts and updates. Index maintenance is deferred until the end of each DML operation. For example, if you insert 1000 rows, the inserted rows are placed into a sort buffer and then the updates of all 1000 index entries are batched. (This is why SORT_AREA_SIZE must be set properly for good performance with inserts and updates on bitmap indexes.) Thus each bitmap segment is updated only once per DML operation, even if more than one row in that segment changes.

Note: The sorts described above are regular sorts and use the regular sort area, determined by SORT_AREA_SIZE. The BITMAP_MERGE_AREA_SIZE and CREATE_BITMAP_AREA_SIZE parameters described in "Initialization Parameters for Bitmap Indexing" on page 6-18 only affect the specific operations indicated by the parameter names.

DML and DDL statements such as UPDATE, DELETE, DROP TABLE, affect bitmap indexes the same way they do traditional indexes: the consistency model is the same. A compressed bitmap for a key value is made up of one or more bitmap segments, each of which is at most half a block in size (but may be smaller). The locking granularity is one such bitmap segment. This may affect performance in environments where many transactions make simultaneous updates. If numerous DML operations have caused increased index size and decreasing performance for queries, you can use the ALTER INDEX ... REBUILD statement to compact the index and restore efficient performance.

A B*-tree index entry contains a single rowid. Therefore, when the index entry is locked, a single row is locked. With bitmap indexes, an entry can potentially contain a range of rowids. When a bitmap index entry is locked, the entire range of rowids is locked. The number of rowids in this range affects concurrency. For example, a bitmap index on a column with unique values would lock one rowid per value: concurrency would be the same as for B*-tree indexes. As the number of rowids increases in a bitmap segment, concurrency decreases.

Locking issues affect DML operations, and thus may affect heavy OLTP environments. Locking issues do not, however, affect query performance. As with other types of indexes, updating bitmap indexes is a costly operation. Nonetheless, for bulk inserts and updates where many rows are inserted or many updates are made in a single statement, performance with bitmap indexes can be better than with regular B*-tree indexes.

Creating Bitmap Indexes

To create a bitmap index, use the BITMAP keyword in the CREATE INDEX statement:

```
CREATE BITMAP INDEX ...
```

Multi-column (concatenated) bitmap indexes are supported; they can be defined over no more than 32 columns. Other SQL statements concerning indexes, such as DROP, ANALYZE, ALTER, and so on, can refer to bitmap indexes without any extra keyword. For information on bitmap index restrictions, please refer to *Oracle8i SQL Reference*

Note: The COMPATIBLE initialization parameter must be set to 7.3.2 or higher to use bitmap indexes.

Index Type

System index views USER_INDEXES, ALL_INDEXES, and DBA_INDEXES indicate bitmap indexes by the word BITMAP appearing in the TYPE column. A bitmap index cannot be declared as UNIQUE.

Using Hints

The INDEX hint works with bitmap indexes in the same way as with traditional indexes.

The INDEX_COMBINE hint identifies the most cost effective hints for the optimizer. The optimizer recognizes all indexes that can potentially be combined, given the predicates in the WHERE clause. However, it may not be cost effective to use all of them. Oracle recommends using INDEX_COMBINE rather than INDEX for bitmap indexes because it is a more versatile hint.

In deciding which of these hints to use, the optimizer includes non-hinted indexes that appear cost effective as well as indexes named in the hint. If certain indexes are given as arguments for the hint, the optimizer tries to use some combination of those particular bitmap indexes.

If the hint does not name indexes, all indexes are considered hinted. Hence, the optimizer tries to combine as many as is possible given the WHERE clause, without regard to cost effectiveness. The optimizer always tries to use hinted indexes in the plan regardless of whether it considers them cost effective.

See Also: "INDEX_COMBINE" on page 7-46.

Performance and Storage Tips

To obtain optimal performance and disk space usage with bitmap indexes, note the following considerations:

- Large block sizes improve the efficiency of storing, and hence retrieving, bitmap indexes
- To make compressed bitmaps as small as possible, declare NOT NULL constraints on all columns that cannot contain null values
- Fixed-length datatypes are more amenable to a compact bitmap representation than variable length datatypes

See Also: Chapter 13, "Using EXPLAIN PLAN" for information about bitmap EXPLAIN PLAN output.

Efficient Mapping of Bitmaps to Rowids

Use SQL statements with the ALTER TABLE syntax to optimize the mapping of bitmaps to rowids. The MINIMIZE RECORDS_PER_BLOCK clause enables this optimization and the NOMINIMIZE RECORDS_PER_BLOCK clause disables it.

When enabled, Oracle scans the table and determines the maximum number of records in any block and restricts this table to this maximum number. This enables bitmap indexes to allocate fewer bits per block and results in smaller bitmap indexes. The block and record allocation restrictions this statement places on the table are only beneficial to bitmap indexes. Therefore, Oracle does not recommend using this mapping on tables that are not heavily indexed with bitmap indexes.

See Also: "Using Bitmap Indexes" on page 6-13. Also refer to the *Oracle8i SQL Reference* for details on the use of the MINIMIZE and NOMINIMIZE syntax.

Indexing Null Values

Bitmap indexes index nulls, whereas all other index types do not. Consider, for example, a table with STATE and PARTY columns, on which you want to perform the following query:

```
SELECT COUNT(*) FROM people WHERE state='CA' and party !='D';
```

Indexing nulls enables a bitmap minus plan where bitmaps for party equal to 'D' and NULL are subtracted from state bitmaps equal to 'CA'. The EXPLAIN PLAN output would look like this:

```
 SELECT STATEMENT
SORT                    AGGREGATE
  BITMAP CONVERSION     COUNT
    BITMAP MINUS
      BITMAP MINUS
        BITMAP INDEX    SINGLE VALUE    STATE_BM
        BITMAP INDEX    SINGLE VALUE    PARTY_BM
      BITMAP INDEX      SINGLE VALUE    PARTY_BM
```

If a NOT NULL constraint existed on party, the second minus operation (where party is null) would be left out because it is not needed.

Initialization Parameters for Bitmap Indexing

The following two initialization parameters have an effect on performance.

CREATE_BITMAP_AREA_SIZE

This parameter determines the amount of memory allocated for bitmap creation. The default value is 8MB. A larger value may lead to faster index creation. If cardinality is very small, you can set a small value for this parameter. For example, if cardinality is only 2, then the value can be on the order of kilobytes rather than megabytes. As a general rule, the higher the cardinality, the more memory is needed for optimal performance. You cannot dynamically alter this parameter at the system or session level.

BITMAP_MERGE_AREA_SIZE

This parameter determines the amount of memory used to merge bitmaps retrieved from a range scan of the index. The default value is 1 MB. A larger value should improve performance because the bitmap segments must be sorted before being merged into a single bitmap. You cannot dynamically alter this parameter at the system or session level.

> **See Also:** For more information on improving bitmap index efficiency, please see "Efficient Mapping of Bitmaps to Rowids" on page 6-17.

Using Bitmap Access Plans on Regular B*-tree Indexes

If there is at least one bitmap index on the table, the optimizer considers using a bitmap access path using regular B*-tree indexes for that table. This access path may involve combinations of B*-tree and bitmap indexes, but might not involve any bitmap indexes at all. However, the optimizer will not generate a bitmap access path using a single B*-tree index unless instructed to do so by a hint.

To use bitmap access paths for B*-tree indexes, the rowids stored in the indexes must be converted to bitmaps. After such a conversion, the various Boolean operations available for bitmaps can be used. As an example, consider the following query, where there is a bitmap index on column C1, and regular B*-tree indexes on columns C2 and C3.

```
EXPLAIN PLAN FOR
SELECT COUNT(*) FROM T
WHERE
C1 = 2 AND C2 = 6
OR
C3 BETWEEN 10 AND 20;
SELECT STATEMENT
  SORT AGGREGATE
    BITMAP CONVERSION COUNT
      BITMAP OR
        BITMAP AND
          BITMAP INDEX  C1_IND  SINGLE VALUE
          BITMAP CONVERSION  FROM ROWIDS
            INDEX  C2_IND  RANGE SCAN
        BITMAP CONVERSION  FROM ROWIDS
          SORT  ORDER BY
            INDEX  C3_IND  RANGE SCAN
```

Here, a COUNT option for the BITMAP CONVERSION row source counts the number of rows matching the query. There are also conversions FROM rowids in the plan to generate bitmaps from the rowids retrieved from the B*-tree indexes. The occurrence of the ORDER BY sort in the plan is due to the fact that the conditions on column C3 result in more than one list of rowids being returned from the B*-tree index. These lists are sorted before they can be converted into a bitmap.

Estimating Bitmap Index Size

Although it is not possible to precisely size a bitmap index, you can estimate its size. This section describes how to determine the size of a bitmap index for a table using the computed size of a B*-tree index. It also illustrates how cardinality, NOT NULL constraints, and number of distinct values affect bitmap size.

To estimate the size of a bitmap index for a given table, you may extrapolate from the size of a B*-tree index for the table. Use the following approach:

1. Use the standard formula described in Oracle8i Concepts to compute the size of a B*-tree index for the table.
2. Determine the cardinality of the table data.
3. From the cardinality value, extrapolate the size of a bitmap index according to the graph in Figure 6–2 or Figure 6–3.

For a 1 million row table, Figure 6–2 shows index size on columns with different numbers of distinct values, for B*-tree indexes and bitmap indexes. Using Figure 6–2 you can estimate the size of a bitmap index relative to that of a B*-tree index for the table. Sizing is not exact: results vary somewhat from table to table.

Randomly distributed data was used to generate the graph. If, in your data, particular values tend to cluster close together, you may generate considerably smaller bitmap indexes than indicated by the graph. Bitmap indexes may be slightly smaller than those in the graph if columns contain NOT NULL constraints.

Figure 6–3 shows similar data for a table with 5 million rows. When cardinality exceeds 100,000, bitmap index size does not increase as fast as it does in Figure 6–2. For a table with more rows, there are more repeating values for a given cardinality.

Figure 6–2 Extrapolating Bitmap Index Size: 1 Million Row Table

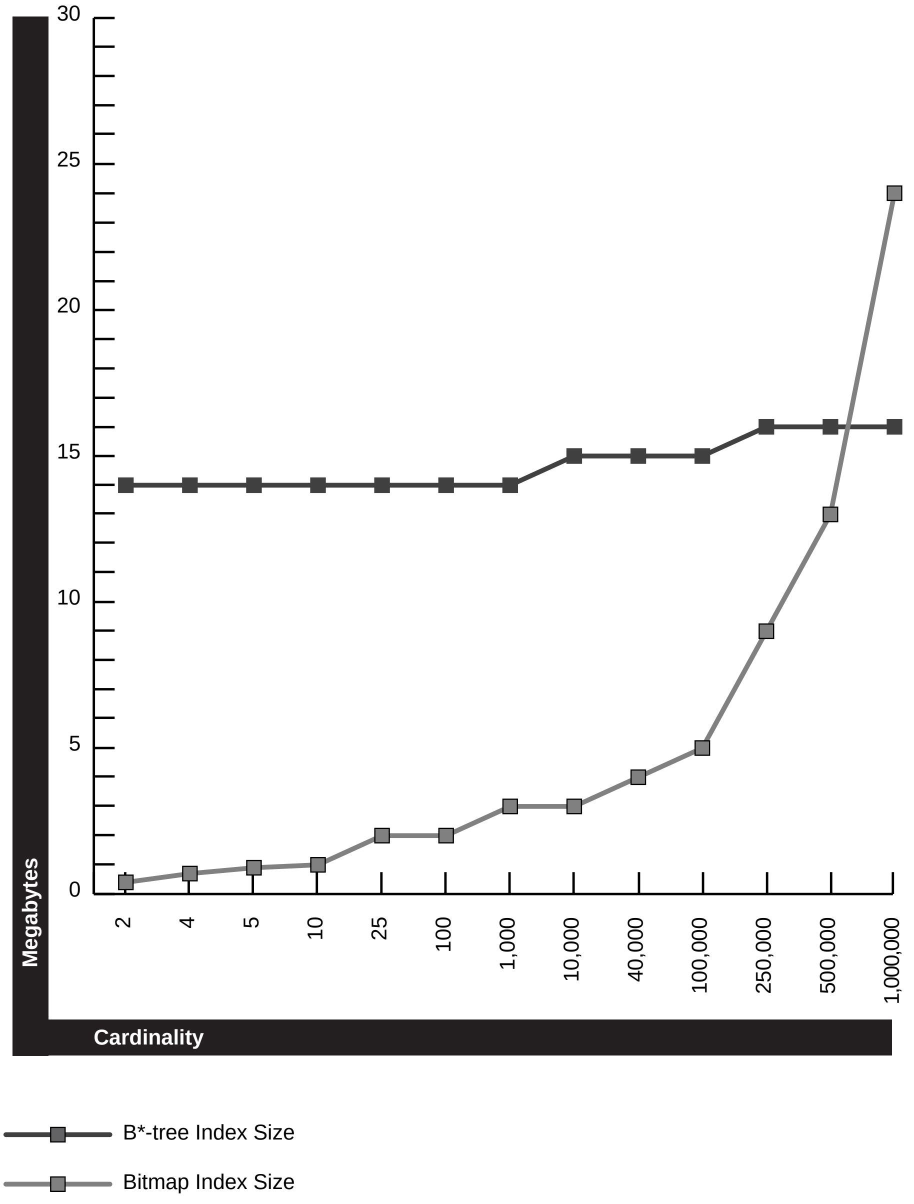
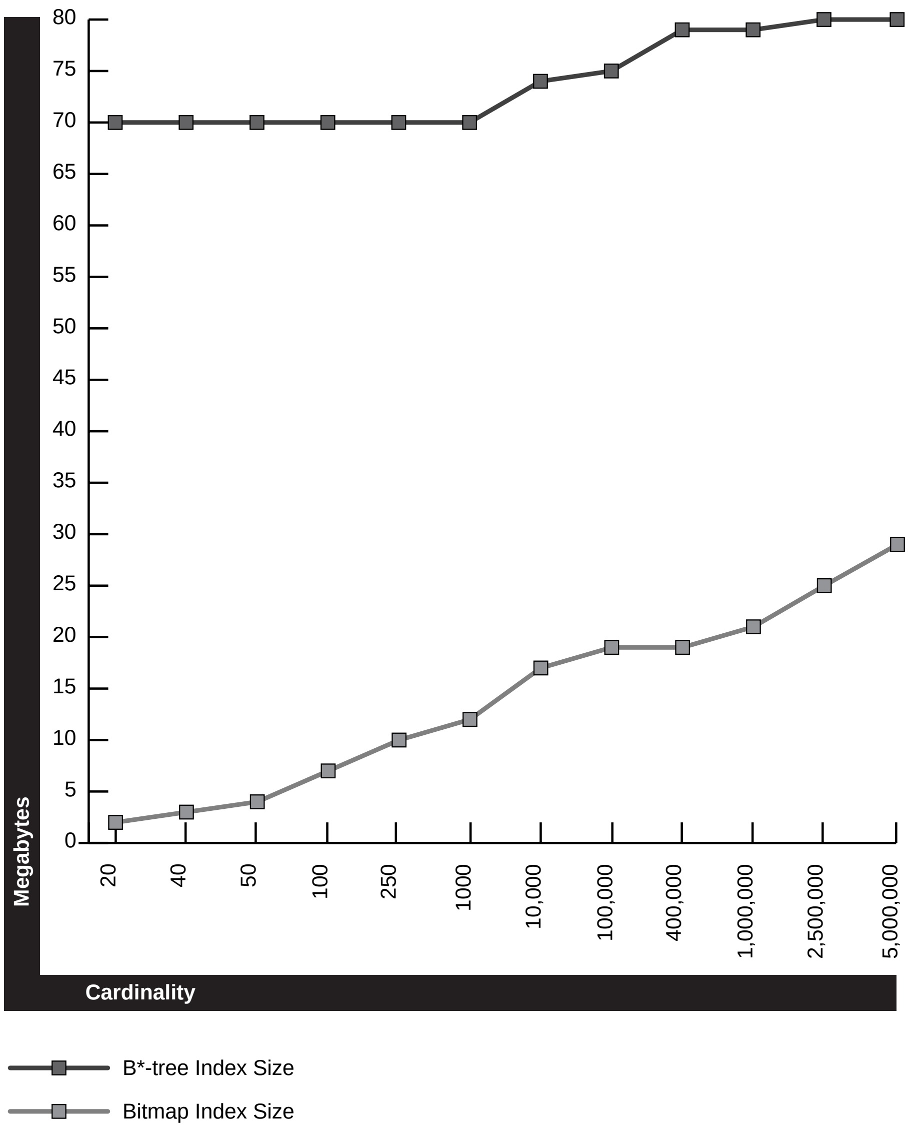


Figure 6–3 Extrapolating Bitmap Index Size: 5 Million Row Table



Bitmap Index Restrictions

Bitmap indexes have the following restrictions:

- For bitmap indexes with direct load, the SORTED_INDEX flag does not apply
- Bitmap indexes are not considered by the rule-based optimizer
- Bitmap indexes cannot be used for referential integrity checking

Using Domain Indexes

Domain indexes are built using the indexing logic supplied by a user-defined indextype. Typically, the user-defined indextype is part of a data cartridge. For example, the Spatial cartridge provides a SpatialIndextype to index spatial data.

An indextype provides an efficient mechanism to access data that satisfy certain operator predicates. For example, the SpatialIndextype allows efficient search and retrieval of spatial data that overlap a given bounding box.

The cartridge determines the parameters you can specify in creating and maintaining the domain index. Similarly, the performance and storage characteristics of the domain index are presented in the specific cartridge documentation.

Refer to the appropriate cartridge documentation for information such as:

- What datatypes can be indexed?
- What indextypes are provided?
- What operators does the indextype support?
- How can the domain index be created and maintained?
- How do we efficiently use the operator in queries?
- What are the performance characteristics?

Note: You can also create index types with the CREATE INDEXTYPE SQL statement.

See Also: For information about the SpatialIndextype, please refer to the *Oracle8i Spatial User's Guide and Reference*.

Using Clusters

Follow these guidelines when deciding whether to cluster tables:

- Consider clustering tables that are often accessed by your application in join statements.
- Do not cluster tables if your application joins them only occasionally or modifies their common column values frequently. Modifying a row's cluster key value takes longer than modifying the value in an unclustered table, because Oracle may have to migrate the modified row to another block to maintain the cluster.
- Do not cluster tables if your application often performs full table scans of only one of the tables. A full table scan of a clustered table can take longer than a full table scan of an unclustered table. Oracle is likely to read more blocks because the tables are stored together.
- Consider clustering 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.
- Consider storing a detail table alone in a cluster if you often select many detail records of the same master. This measure improves the performance of queries that select detail records of the same master but does not decrease the performance of a full table scan on the master table.
- Do not cluster tables if the data from all tables with the same cluster key value exceeds more than one or two Oracle blocks. To access a row in a clustered table, Oracle reads all blocks containing rows with that value. If these rows take up multiple blocks, accessing a single row could require more reads than accessing the same row in an unclustered table.

Consider the benefits and drawbacks of clusters with respect to the needs of your application. For example, you may decide that the performance gain for join statements outweighs the performance loss for statements that modify cluster key values. You may want to experiment and compare processing times with your tables both clustered and stored separately. To create a cluster, use the CREATE CLUSTER statement.

See Also: For more information on creating clusters, see *Oracle8i Application Developer's Guide - Fundamentals.*

Using Hash Clusters

Hash clusters group table data by applying a hash function to each row's cluster key value. All rows with the same cluster key value are stored on disk. Consider the benefits and drawbacks of hash clusters with respect to the needs of your application. You may want to experiment and compare processing times with a particular table as it is stored in a hash cluster, and as it is stored alone with an index. This section describes:

- When to Use a Hash Cluster
- Using Hash Clusters

When to Use a Hash Cluster

Follow these guidelines for choosing when to use hash clusters:

- Consider using hash clusters to store tables often accessed by SQL statements with WHERE clauses if the WHERE clauses contain equality conditions that use the same column or combination of columns. Designate this column or combination of columns as the cluster key.
- Store a table in a hash cluster if you can determine how much space is required to hold all rows with a given cluster key value, including rows to be inserted immediately as well as rows to be inserted in the future.
- Do not use hash clusters if space in your database is scarce and you cannot afford to allocate additional space for rows to be inserted in the future.
- Do not use a hash cluster to store a constantly growing table if the process of occasionally creating a new, larger hash cluster to hold that table is impractical.
- Do not store a table in a hash cluster if your application often performs full table scans and you must allocate a great deal of space to the hash cluster in anticipation of the table growing. Such full table scans must read all blocks allocated to the hash cluster, even though some blocks may contain few rows. Storing the table alone would reduce the number of blocks read by full table scans.
- Do not store a table in a hash cluster if your application frequently modifies the cluster key values. Modifying a row's cluster key value can take longer than modifying the value in an unclustered table, because Oracle may have to migrate the modified row to another block to maintain the cluster.

- Storing a single table in a hash cluster can be useful, regardless of whether the table is often joined with other tables, provided that hashing is appropriate for the table based on the previous points in this list.

Creating Hash Clusters

To create a hash cluster, use the CREATE CLUSTER statement with the HASHKEYS parameter.

When you create a hash cluster, you must use the HASHKEYS parameter of the CREATE CLUSTER statement to specify the number of hash values for the hash cluster. For best performance of hash scans, choose a HASHKEYS value that is at least as large as the number of cluster key values. Such a value reduces the chance of collisions, or multiple cluster key values resulting in the same hash value. Collisions force Oracle to test the rows in each block for the correct cluster key value after performing a hash scan. Collisions reduce the performance of hash scans.

Oracle always rounds up the HASHKEYS value that you specify to the nearest prime number to obtain the actual number of hash values. This rounding is designed to reduce collisions.

> **See Also:** For more information on creating hash clusters, see *Oracle8i Application Developer's Guide - Fundamentals.*

7

Optimizer Modes, Plan Stability, and Hints

This chapter explains cost-based and rule-based optimization. It also describes how to use Plan Stability to preserve performance characteristics and to migrate to the cost-based optimizer. It also describes using hints to enhance Oracle performance. Topics in this chapter include:

See Also: *Oracle8i Concepts* for an introduction to the optimizer, access methods, join operations, and parallel execution.

Using Cost-based Optimization

This section discusses:

- When to Use the Cost-based Approach
- Using the Cost-based Approach
- Choosing a Goal for the Cost-based Approach
- Using Histograms for Nonuniformly Distributed Data
- Generating Statistics
- Automated Statistics Gathering
- Parameters Affecting Cost-based Optimization Plans
- Parameters Affecting How the Optimizer Uses Indexes
- Tips for Using the Cost-based Approach

This section also includes a brief discussion on:

- Using Rule-Based Optimization

When to Use the Cost-based Approach

In general, always use the cost-based optimization approach. The rule-based approach is available for the benefit of existing applications, but new optimizer functionality uses the cost-based approach.

The following features are available *only* with cost-based optimization.

- Partitioned tables
- Index-organized tables
- Reverse indexes
- Parallel execution
- Star transformations
- Star joins

Note: You *must* gather statistics for your tables to obtain accurate execution plans.

The cost-based approach generally chooses an execution plan that is as good as or better than the plan chosen by the rule-based approach. This is specially true for large queries with multiple joins or multiple indexes. The cost-based approach also eliminates having to tune your SQL statements; this greatly improves productivity.

Use cost-based optimization for efficient star query performance. Similarly, use it with hash joins and histograms. Cost-based optimization is always used with parallel execution and with partitioned tables. To maintain the effectiveness of the cost-based optimizer, you must keep statistics current.

See Also: For information on moving from the rule-based optimizer to the cost-based optimizer, refer to "Using Outlines to Move to the Cost-based Optimizer" on page 7-32.

Using the Cost-based Approach

To use cost-based optimization for a statement, collect statistics for the tables accessed by the statement and enable cost-based optimization using one of these methods:

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

The plans generated by the cost-based optimizer depend on the sizes of the tables. When using the cost-based optimizer with a small amount of data to test an application prototype, do not assume the plan chosen for the full-size database will be the same as that chosen for the prototype.

See Also: For information on upgrading to more recent cost-based optimizer versions, please refer to "RDBMS Upgrades and the Cost-based Optimizer" on page 7-34.

Choosing a Goal for the Cost-based Approach

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

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

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

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

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

- To change the goal of the cost-based approach for all SQL statements in your session, issue an ALTER SESSION SET OPTIMIZER_MODE statement with the ALL_ROWS or FIRST_ROWS option.
- To specify the goal of the cost-based approach for an individual SQL statement, use the ALL_ROWS or FIRST_ROWS hint.

Example: This statement changes the goal of the cost-based approach for your session to best response time:

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

Using Histograms for Nonuniformly Distributed Data

For uniformly distributed data, the cost-based approach fairly accurately determines the cost of executing a particular statement. In such cases, the optimizer does not need histograms to estimate the cost of a query.

However, for nonuniformly distributed data, Oracle allows you to create histograms that describe data distribution patterns of a particular column. Oracle stores these histograms in the data dictionary for use by the cost-based optimizer.

Histograms are persistent objects, so you incur maintenance and space costs for using them. Compute histograms only for columns with highly skewed data distributions. The statistics that Oracle uses to build histograms, as well as all optimizer statistics, are static. If the data distribution of a column changes frequently, gather new histogram statistics for that column. Do this either explicitly, or by using the Automated Statistics Gathering feature as described on page 7-19.

Histograms are not useful for columns with these characteristics:

- All predicates on the column use bind variables
- The column data is uniformly distributed
- The column is not used in WHERE clauses of queries
- The column is unique and is used only with equality predicates

Creating Histograms

Create histograms on columns that are frequently used in WHERE clauses of queries and that have highly skewed data distributions. To do this, use the GATHER_TABLE_STATS procedure of the DBMS_STATS package. For example, to create a 10-bucket histogram on the SAL column of the EMP table, issue this statement:

```
EXECUTE DBMS_STATS.GATHER_TABLE_STATS
('scott','emp', METHOD_OPT => 'FOR COLUMNS SIZE 10 sal');
```

The SIZE keyword declares the maximum number of buckets for the histogram. You would create a histogram on the SAL column if there was an unusually high number of employees with the same salary and few employees with other salaries. You can also collect histograms for a single partition of a table.

Column statistics appear in the data dictionary views: USER_TAB_COLUMNS, ALL_TAB_COLUMNS, and DBA_TAB_COLUMNS.

Histograms appear in the data dictionary views USER_HISTOGRAMS, DBA_HISTOGRAMS, and ALL_HISTOGRAMS.

> **See Also:** For more information on the DBMS_STATS package, refer to "Gathering Statistics with the DBMS_STATS Package" on page 7-9. For more information about the ANALYZE statement and its options, refer to the *Oracle8i SQL Reference*.

Choosing the Number of Buckets for a Histogram

The default number of buckets for a histogram is 75. This value provides an appropriate level of detail for most data distributions. However, since the number of buckets in the histogram, also referred to as 'the sampling rate', and the data distribution all affect a histogram's usefulness, you may need to experiment with different numbers of buckets to obtain optimal results.

If the number of frequently occurring distinct values in a column is relatively small, set the number of buckets to be greater than the number of frequently occurring distinct values.

Viewing Histograms

You can find information about existing histograms in the database using these data dictionary views:

- USER_HISTOGRAMS
- ALL_HISTOGRAMS
- DBA_HISTOGRAMS

Find the number of buckets in each column's histogram in:

- USER_TAB_COLUMNS
- ALL_TAB_COLUMNS
- DBA_TAB_COLUMNS

> **See Also:** *Oracle8i Reference* for column descriptions of data dictionary views as well as histogram use and restrictions.

Generating Statistics

Because the cost-based approach relies on statistics, generate statistics for all tables, clusters, and all types of indexes accessed by your SQL statements before using the cost-based approach. If the size and data distribution of these tables change frequently, generate these statistics regularly to ensure the statistics accurately represent the data in the tables.

Oracle can generate statistics using these techniques:

- Estimation based on random data sampling
- Exact computation
- Using user-defined statistics collection methods

Because of the time and space required for computing table statistics, use estimation for tables and clusters rather than computation unless you need exact values. The reasons for this are:

- Exact computation always provides exact values but can take longer than estimation. The time necessary to compute statistics for a table is approximately the time required to perform a full table scan and a sort of the rows in the table.
- Estimation is often much faster than computation, especially for large tables, because estimation never scans the entire table.

To perform an exact computation, Oracle requires enough space to perform a scan and sort of the table. If there is not enough space in memory, temporary space may be required. For estimations, Oracle requires enough space to perform a scan and sort of all rows in the requested sample of the table. For indexes, computation does not take up as much time or space, so it is best to perform a full computation.

When you generate statistics for a table, column, or index, if the data dictionary already contains statistics for the analyzed object, Oracle updates the existing statistics. Oracle also invalidates any currently parsed SQL statements that access any of the analyzed objects.

The next time such a statement executes, the optimizer automatically chooses a new execution plan based on the new statistics. Distributed statements issued on remote databases that access the analyzed objects use the new statistics the next time Oracle parses them.

Some statistics are always computed, regardless of whether you specify computation or estimation. If you specify estimation and the time saved by estimating statistics is negligible, Oracle computes the statistics.

When you associate a statistics type with a column or domain index, Oracle calls the statistics collection method in the statistics type if you analyze the column or domain index.

> **See Also:** For more information about user-defined statistics, please refer to the *Oracle8i Data Cartridge Developer's Guide*. For more information about the ANALYZE statement, please refer to the *Oracle8i SQL Reference.*

Gather statistics with the DBMS_STATS package as explained under the following heading.

Gathering Statistics with the DBMS_STATS Package

DBMS_STATS provides the following procedures for gathering statistics:

Table 7–1 Statistics Gathering Procedures in the DBMS_STATS Package

Procedure	Description
GATHER_ INDEX_STATS	Collects index statistics.
GATHER_TABLE_STATS	Collects table, column, and index statistics.
GATHER_SCHEMA_STATS	Collects statistics for all objects in a schema.
GATHER_DATABASE_STATS	Collects statistics for all objects in a database.

Gathering Index Statistics

Use the COMPUTE STATISTICS clause of the ANALYZE SQL statement to gather index statistics when creating or rebuilding an index. If you do not use the COMPUTE STATISTICS clause, or you have made major DML changes, use the GATHER_INDEX_STATS procedure to collect index statistics. The GATHER_INDEX_STATS procedure does not run in parallel. Using this procedure is equivalent to running:

```
ANALYZE INDEX [ownname.]indname [PARTITION partname] COMPUTE STATISTICS |
ESTIMATE STATISTICS SAMPLE estimate_percent PERCENT
```

See Also: For more information about the COMPUTE STATISTICS clause, please refer to the *Oracle8i SQL Reference*.

Example

This PL/SQL example gathers index statistics:

```
EXECUTE DBMS_STATS.GATHER_INDEX_STATS(
  'scott','emp_idx');
```

Syntax

The syntax and parameters for the GATHER_INDEX_STATS procedure are:

```
PROCEDURE GATHER_INDEX_STATS(
  ownname VARCHAR2,
  indname VARCHAR2,
  partname VARCHAR2 DEFAULT NULL,
  estimate_percent NUMBER DEFAULT NULL,
  stattab VARCHAR2 DEFAULT NULL,
  statid VARCHAR2 DEFAULT NULL,
  statown VARCHAR2 DEFAULT NULL);
```

ownname	Schema of the index to analyze.
indname	Name of index.
partname	Name of partition.
estimate_percent	Percentage of rows to estimate (NULL means "compute"). The valid range is [0.000001,100).
stattab	Name of a user statistics table into which Oracle should back up original statistics before collecting new statistics. For more information, please see the *Oracle8i Supplied Packages Reference*.
statid	Secondary identifier in stattab for backing up statistics.
statown	Schema where Oracle stores stattab if different from the location identified by the ownname parameter.

Gathering Table Statistics

After creating a table, use the GATHER_TABLE_STATS procedure to collect table, column, and index statistics as shown in the following example. The GATHER_TABLE_STATS procedure uses parallel processing for as much of the process as possible.

Example

This PL/SQL example gathers table statistics:

```
EXECUTE DBMS_STATS.GATHER_TABLE_STATS (
  ’scott’,’emp’);
```

Syntax

The syntax and parameters for the GATHER_TABLE_STATS procedure are:

```
PROCEDURE GATHER_TABLE_STATS
  ownname VARCHAR2,
  tabname VARCHAR2,
  partname VARCHAR2 DEFAULT NULL,
  estimate_percent NUMBER DEFAULT NULL,
  block_sample BOOLEAN DEFAULT FALSE,
  method_opt VARCHAR2 DEFAULT `FOR ALL COLUMNS SIZE 1',
  degree NUMBER DEFAULT NULL,
  granularity VARCHAR2 DEFAULT `DEFAULT',
  cascade BOOLEAN DEFAULT FALSE,
  stattab VARCHAR2 DEFAULT NULL,
  statid VARCHAR2 DEFAULT NULL,
  statown VARCHAR2 DEFAULT NULL);
```

ownname	Name of the schema of table to analyze.
tabname	Name of table.
partname	Name of partition.
estimate_percent	Percentage of rows to estimate (NULL means "compute"). The valid range is [0.000001,100).

block_sample	This value indicates whether to use random block sampling instead of the default random row sampling. Random block sampling is more efficient, but if the data is not randomly distributed on disk then the sample values may be somewhat correlated which may lead to less-than-accurate statistics. Only pertinent when estimating statistics. For more information about BLOCK_SAMPLE, please refer to *Oracle8i Concepts*.
method_opt	This value holds the options of the following format: (The phrase 'SIZE 1' (in other words, no histograms) is required to ensure gathering statistics in parallel): FOR ALL [INDEXED] COLUMNS [SIZE integer] FOR COLUMNS [SIZE integer] column | attribute [column | attribute...] Optimizer-related table statistics are always gathered.
degree	This value specifies the degree of parallelism (NULL means "use default table value").
granularity	This value determines the granularity of statistics to collect (this only affects partitioned tables). PARTITION - Gather partition-level statistics. GLOBAL - Gather global statistics. ALL - Gather all statistics. SUBPARTITION - Gather subpartition level statistics. DEFAULT - Gather partition and global statistics.
cascade	Setting this parameter to TRUE gathers statistics on the indexes for this table. Index statistics are not gathered in parallel. Using this option is equivalent to running the GATHER_INDEX_STATS procedure on each of the table's indexes. The default value is FALSE due to the assumption that index statistics were recently gathered using the statement syntax 'CREATE INDEX... COMPUTE STATISTICS'.

stattab	Name of a user statistics table into which Oracle should back up original statistics before collecting new statistics. For more information on stattab and the following parameters, statid and statown, please refer to the *Oracle8i Supplied Packages Reference*.
statid	Secondary identifier in stattab for backing up statistics.
statown	Schema where Oracle stores stattab if different from the location identified by the ownname parameter.

Gathering Schema Statistics

Run the GATHER_SCHEMA_STATS procedure to collect statistics for all objects in a schema as shown in the following example.

Example

This PL/SQL example gathers schema statistics:

```
EXECUTE DBMS_STATS.GATHER_SCHEMA_STATS('scott');
```

Syntax

The syntax and parameters for the GATHER_SCHEMA_STATS procedure are explained here. Oracle passes the values to all tables for all parameters shown.

```
PROCEDURE GATHER_SCHEMA_STATS(
  ownname VARCHAR2,
  estimate_percent NUMBER DEFAULT NULL,
  block_sample BOOLEAN DEFAULT FALSE,
  method_opt VARCHAR2 DEFAULT 'FOR ALL COLUMNS SIZE 1',
  degree NUMBER DEFAULT NULL, INSTANCES NUMBER DEFAULT NULL,
  granularity VARCHAR2 DEFAULT 'ALL',
  cascade BOOLEAN DEFAULT FALSE,
  stattab VARCHAR2 DEFAULT NULL,
  statid VARCHAR2 DEFAULT NULL,
  options VARCHAR2 DEFAULT 'GATHER',
  objlist OUT OBJECTTAB,
  statown VARCHAR2 DEFAULT NULL);
```

ownname	The name of the schema to analyze (NULL means "current user's schema").
estimate_percent	The percentage of rows to estimate (NULL means "compute") The valid range is [0.000001,100).
block_sample	Indicates whether to use random block sampling instead of the default random row sampling. Random block sampling is more efficient, but if the data is not randomly distributed on disk then the sample values may be somewhat correlated which may lead to less-than-accurate statistics. Only pertinent when doing an estimate of statistics.

method_opt	Indicates the method options of the following format (the phrase ’SIZE 1’ (in other words, no histograms) is required to enable gathering of statistics in parallel): FOR ALL [INDEXED] COLUMNS [SIZE integer].
degree	Shows the degree of parallelism (NULL means "use table default value").
granularity	Shows the granularity of statistics to collect. This only applies to partitioned tables. PARTITION - Gather partition-level statistics. GLOBAL - Gather global statistics. ALL - Gather all statistics. SUBPARTITION - Gather subpartition level statistics. DEFAULT - Gather partition and global statistics.
cascade	Shows to gather statistics on the indexes as well. Index statistics gathering will not be parallelized. Using this option is equivalent to running the GATHER_INDEX_STATS procedure on each of the indexes in the schema in addition to gathering table and column statistics.
stattab	Name of a user statistics table into which Oracle should back up original statistics before collecting new statistics. For more information, please see the *Oracle8i Supplied Packages Reference*.
statid	Secondary identifier in stattab for backing up statistics. For more information, please see the *Oracle8i Supplied Packages Reference*.
options	Please refer to the following section Automated Statistics Gathering on page 7-19.
objlist	Please refer to the following section Automated Statistics Gathering on page 7-19.
statown	Schema where Oracle stores stattab if different from the location identified by the ownname parameter. For more information, please see the *Oracle8i Supplied Packages Reference*.

Gathering Database Statistics

After creating your database, run the GATHER_DATABASE_STATS procedure to collect database statistics.

Example

This PL/SQL example gathers database statistics:

```
EXECUTE DBMS_STATS.GATHER_DATABASE_STATS;
```

Syntax

The syntax and parameters for the GATHER_DATABASE_STATS procedure are:

```
PROCEDURE GATHER_DATABASE_STATS(
  estimate_percent NUMBER DEFAULT NULL,
  block_sample BOOLEAN DEFAULT FALSE,
  method_opt VARCHAR2 DEFAULT `FOR ALL COLUMNS SIZE 1',
  degree NUMBER DEFAULT NULL,
  granularity VARCHAR2 DEFAULT,
  cascade BOOLEAN DEFAULT FALSE,
  stattab VARCHAR2 DEFAULT NULL,
  statid VARCHAR2 DEFAULT NULL,
  options VARCHAR2 DEFAULT ’GATHER’,
  objlist OUT OBJECTTAB,
  statown VARCHAR2 DEFAULT NULL);
```

estimate_percent	Shows the percentage of rows to estimate (NULL means "compute"). The valid range is [0.000001,100).
block_sample	Shows whether to use random block sampling instead of the default random row sampling. Random block sampling is more efficient, but if the data is not randomly distributed on disk then the sample values may be somewhat correlated which may lead to less-than-accurate statistics. Only applies when doing an estimate of statistics.
method_opt	Indicates the method options of the following format (the phrase ’SIZE 1’ (in other words, no histograms) is required to enable gathering of statistics in parallel): FOR ALL [INDEXED] COLUMNS [SIZE integer].
degree	Shows the degree of parallelism (NULL means "use table default value").

granularity	Indicates the granularity of statistics to collect (applies only to partitioned tables). PARTITION - Gather partition-level statistics. GLOBAL - Gather global statistics. ALL - Gather all statistics. SUBPARTITION - Gather subpartition level statistics. DEFAULT - Gather partition and global statistics.
cascade	Setting this value to TRUE gathers statistics on the indexes as well. Index statistics gathering is not parallelized. Using this option is equivalent to running the GATHER_INDEX_STATS procedure on all indexes in the database in addition to gathering table and column statistics.
stattab	Name of a user statistics table into which Oracle should back up original statistics before collecting new statistics. For more information, please see the *Oracle8i Supplied Packages Reference*.
statid	Secondary identifier in stattab for backing up statistics. For more information, please see the *Oracle8i Supplied Packages Reference*.
options	Please refer to the following section "Automated Statistics Gathering" on page 7-19.
objlist	Please refer to the following section "Automated Statistics Gathering" on page 7-19.
statown	Schema where Oracle stores stattab if different from the location identified by the ownname parameter. For more information, please see the *Oracle8i Supplied Packages Reference*.

Gathering New Optimizer Statistics

Oracle recommends the following procedure for gathering new optimizer statistics for a particular schema.

Before gathering new statistics, use DBMS_STATS.EXPORT_SCHEMA_STATS to extract and save existing statistics. Then use DBMS_STATS.GATHER_SCHEMA_STATS to gather new statistics. You can implement both of these with a single call to the GATHER_SCHEMA_STATS procedure.

If key SQL statements experience significant performance degradation, either gather statistics again using a larger sample size, or perform these steps:

1. Use the DBMS_STATS.EXPORT_SCHEMA_STATS procedure to save the new statistics.
2. Use DBMS_STATS.IMPORT_SCHEMA_STATS to restore the old statistics. The application is now ready to run again.

You may want to use the new statistics if they result in improved performance for the majority of SQL statements, and if the number of problem SQL statements is small. In this case, do the following:

1. Create a stored outline for each problematic SQL statement using the old statistics.

   Note: Stored outlines are pre-compiled execution plans that Oracle can use to mimic proven application performance characteristics.

2. Use DBMS_STATS.IMPORT_SCHEMA_STATS to restore the new statistics.
3. Your application is now ready to run with the new statistics. However, you will continue to achieve the previous performance levels for the problem SQL statements.

See Also: For more information on stored outlines, please refer to "Creating Outlines" on page 7-28.

Automated Statistics Gathering

This feature allows you to automatically gather statistics. You can also use it to create lists of tables that have stale statistics or to create lists of tables that have no statistics.

Use this feature by running the GATHER_SCHEMA_STATS and GATHER_DATABASE_STATS procedures with the *options* and *objlist* parameters. Use the following values for the *options* parameter:

- GATHER STALE - to gather statistics on tables with stale statistics
- GATHER - (default) to gather statistics on all tables
- GATHER EMPTY - to gather statistics only on tables without statistics
- LIST STALE - to create a list of tables with stale statistics
- LIST EMPTY - to create a list of tables that do not have statistics

The *objlist* parameter identifies an output parameter for the LIST STALE and LIST EMPTY options. The *objlist* parameter is of type DBMS_STATS.OBJECTTAB.

Enabling Automated Statistics Gathering

The GATHER STALE option only gathers statistics for tables that have stale statistics and for which you have enabled the MONITORING attribute. To enable monitoring for tables, use the MONITORING keyword of the CREATE TABLE and ALTER TABLE statements as described later in this chapter under the heading "Designating Tables for Monitoring and Automated Statistics Gathering" on page 7-20.

The GATHER STALE option maintains up-to-date statistics for the cost-based optimizer. Using this option at regular intervals also avoids the overhead associated with using the ANALYZE statement on all tables at one time. Using the GATHER option can incur significantly greater overhead since this option will likely gather statistics for a greater number of tables than GATHER STALE.

Use a script or job scheduling tool for the GATHER_SCHEMA_STATS and GATHER_DATABASE_STATS procedures to establish a frequency of statistics collection that is appropriate for your application. The frequency of collection intervals should balance the task of providing accurate statistics for the optimizer against the processing overhead incurred by the statistics collection process.

Creating Lists of Tables with Stale or No Statistics

You can use the GATHER_SCHEMA_STATS and GATHER_DATABASE_STATS procedures to create a list of tables with stale statistics. Use this list to identify tables for which you want to manually gather statistics.

You can also use these procedures to create a list of tables with no statistics. Use this list to identify tables for which you want to gather statistics, either automatically or manually.

Designating Tables for Monitoring and Automated Statistics Gathering

To automatically gather statistics for a particular table, enable the monitoring attribute using the MONITORING keyword. This keyword is part of the CREATE TABLE and ALTER TABLE statement syntax.

Once enabled, Oracle monitors the table for DML activity. This includes the approximate number of inserts, updates, and deletes for that table since the last time statistics were gathered. Oracle uses this data to identify tables with stale statistics.

View the data Oracle obtains from monitoring these tables by querying the USER_TAB_MODIFICATIONS view.

Note: There may be a few hours' delay while Oracle propagates information to this view.

To disable monitoring of a table, use the NOMONITORING keyword.

See Also: For more information about the CREATE TABLE and ALTER TABLE syntax and the MONITORING and NOMONITORING keywords, please refer to the *Oracle8i SQL Reference*.

Preserving Versions of Statistics

You can preserve versions of statistics for tables by specifying the *stattab*, *statid*, and *statown* parameters. Use *stattab* to identify a destination table for archiving previous versions of statistics. Further identify these versions using *statid*, for example, to denote the date and time the version was made. Use *statown* to identify a destination schema if it is different from the schema(s) of the actual tables.

See Also: The parameters *stattab*, *statid*, and *statown* are described in the parameter description lists appearing earlier in this chapter for the GATHER_SCHEMA_STATS and GATHER_DATABASE_STATS procedures.

Parameters Affecting Cost-based Optimization Plans

The following parameters affect cost-based optimization plans:

OPTIMIZER_FEATURES_ENABLED	Turns on a number of optimizer features, including B_TREE_BITMAP_PLANS, and FAST_FULL_SCAN_ENABLED.
OPTIMIZER_MODE	As an initialization parameter, sets the mode of the optimizer at instance startup: rule-based, cost based optimized for throughput or response time, or a choice based on presence of statistics. Set the OPTIMIZER_MODE parameter of the ALTER SESSION statement to change the value dynamically during a session.
OPTIMIZER_PERCENT_PARALLEL	Defines the amount of parallelism that the optimizer uses in its cost functions.
HASH_AREA_SIZE	Larger values can lower hash join costs, thus permitting Oracle to perform more hash joins.
SORT_AREA_SIZE	Larger values can lower sort costs, thus permitting Oracle to perform more sort merge joins.
DB_FILE_MULTIBLOCK_READ_COUNT	Larger values can lower table scan costs and make Oracle favor table scans over indexes.

You often need to set the following parameters in data warehousing applications:

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

You rarely need to change the following parameters:

HASH_MULTIBLOCK_IO_COUNT	Larger value can lower hash join costs thus permitting Oracle to perform more hash joins.
OPTIMIZER_SEARCH_LIMIT	The maximum number of tables in the FROM clause for which all possible join permutations will be considered.
BITMAP_MERGE_AREA_SIZE	The size of the area used to merge the different bitmaps that match a range predicate. Larger size will favor use of bitmap indexes for range predicates.

See Also: *Oracle8i Reference* for complete information about each parameter.

Parameters Affecting How the Optimizer Uses Indexes

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

- OPTIMIZER_INDEX_COST_ADJ
- OPTIMIZER_INDEX_CACHING

Using OPTIMIZER_INDEX_COST_ADJ

Use the OPTIMIZER_INDEX_COST_ADJ parameter to encourage the use of indexes. This parameter encourages the use of all indexes regardless of their selectivity. It also applies to index use in general rather than to just modeling index caching for nested loop join probes.

Using OPTIMIZER_INDEX_CACHING

Use OPTIMIZER_INDEX_CACHING if these two conditions exists:

- If indexes Oracle could use for nested loop join probes are frequently cached in your environment
- If the optimizer is not using nested loop joins aggressively enough

In such an environment, this parameter has two advantages over OPTIMIZER_INDEX_COST_ADJ. First, OPTIMIZER_INDEX_CACHING favors using selective indexes. That is, if you use a relatively low value for this parameter, the optimizer effectively models the caches of all non-leaf index blocks. In this case, the optimizer bases the cost of using this index primarily on the basis of its selectivity. Thus, by setting OPTIMIZER_INDEX_CACHING to a low value, you achieve the desired modeling of the index caching without over using possibly undesirable indexes that have poor selectivity.

Second, the effects of using OPTIMIZER_INDEX_CACHING are restricted to modeling the use of cached indexes for nested loop join probes. Thus, its use has fewer side effects.

Tips for Using the Cost-based Approach

This section describes additional information for using the cost-based approach.

Query Plans for Systems with Large Caches

Cost-based optimization assumes that queries are executed on a multi-user system with fairly low buffer cache hit rates. Thus, a plan selected by the cost-based optimizer may not be the best plan for a single-user system with a large buffer cache. Furthermore, timing a query plan on a single-user system with a large cache may not be a good predictor of performance for the same query on a busy multi-user system.

Analyze Indexes Before Tables

Analyzing a table uses more system resources than analyzing an index. It may be helpful to analyze the indexes for a table separately, or collect statistics during index creation with a higher sampling rate.

Generate Statistics Whenever Possible

Use of access path and join method hints invokes cost-based optimization. Since cost-based optimization is dependent on statistics, it is important to gather statistics for all tables referenced in a query that has hints, even though rule-based optimization may have been selected as the system default.

Provide Statistics Collection, Selectivity, and Cost Functions for User-defined Structures

User-defined structures such as columns, standalone functions, types, packages, indexes, and indextypes, are generally "opaque" to the optimizer. That is, the optimizer does not have statistics about these structures, nor can it compute accurate selectivities or costs for queries that use them.

For this reason, Oracle strongly encourages you to provide statistics collection, selectivity, and cost functions for user-defined structures. This is because the optimizer defaults can be inaccurate and lead to expensive execution plans.

Using Rule-Based Optimization

Oracle supports rule-based optimization, but you should design new applications to use cost-based optimization. You should also use cost-based optimization for data warehousing applications because the cost-based optimizer supports new and enhanced features for DSS.Much of the more recent performance enhancements, such as hash joins, improved star query processing, and histograms, are only available through cost-based optimization.

If you have developed OLTP applications using version 6 of Oracle and have tuned your SQL statements carefully based on the rules of the optimizer, you may want to continue using rule-based optimization when you upgrade these applications to a new version of Oracle.

If you neither collect statistics nor add hints to your SQL statements, your statements will use rule-based optimization. However, you should eventually migrate your existing applications to use the cost-based approach, because eventually, the rule-based approach will not be available in the Oracle server.

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

You can enable cost-based optimization on a trial basis simply by collecting statistics. You can then return to rule-based optimization by deleting the statistics or by setting either the value of the OPTIMIZER_MODE initialization parameter or the

OPTIMIZER_MODE option of the ALTER SESSION statement to RULE. You can also use this value if you want to collect and examine statistics for your data without using the cost-based approach.

See Also: For procedures to migrate from the rule-based optimizer to the cost-based optimizer, refer to "Plan Stability Procedures for the Cost-based Optimizer" on page 7-32. For an explanation of how to gather statistics, please refer to "Gathering Statistics with the DBMS_STATS Package" on page 7-9.

Using Plan Stability to Preserve Execution Plans

Plan Stability prevents certain database environment changes from affecting the performance characteristics of your applications. Such changes include changes to the optimizer mode settings and changes to parameters affecting the sizes of memory structures such as SORT_AREA_SIZE, and BITMAP_MERGE_AREA_SIZE. Plan Stability is most useful when you cannot risk any performance changes in your applications.

Plan Stability preserves execution plans in "stored outlines". Oracle can create a stored outline for one or all SQL statements. The optimizer then generates equivalent execution plans from the outlines when you enable the use of stored outlines.

The plans Oracle maintains in stored outlines remain consistent despite changes to your system’s configuration or statistics. Using stored outlines also stabilizes the generated execution plan if the optimizer changes in subsequent Oracle releases. You can also group outlines into categories and control which category of outlines Oracle uses to simplify outline administration and deployment.

Plan Stability also facilitates migration from the rule-based optimizer to the cost-based optimizer when you upgrade to a new version of Oracle.

Note: If you develop applications for mass distribution, you can use stored outlines to ensure all your customers access the same execution plans. For more information about this, please refer to "Automated Statistics Gathering" on page 7-19.

Plan Stability Uses Hints and Exact Text Matching

The degree to which Plan Stability controls execution plans is dictated by the extent to which Oracle’s hint mechanism controls access paths because Oracle uses hints to record stored plans. Plan Stability also relies on "exact text matching" of queries when determining whether a query has a stored outline.

Similar SQL statements could potentially share stored outlines, however, there is a one-to-one correspondence between SQL text and its stored outline. If you specify a different literal in a predicate, then a different outline applies. To avoid this, replace literals in your applications with bind variables. This gives your SQL statements the exact textual match for outline sharing.

> **See Also:** For more information on how Oracle matches SQL statements to outlines, please refer to the following heading, "Matching SQL Statements with Outlines" on page 7-27.

Plan Stability relies on preserving execution plans at a point in time when performance is satisfactory. In many environments, however, attributes for datatypes such as "dates" or "order numbers" can change rapidly. In these cases, permanent use of an execution plan may result in performance degradation over time as the data characteristics change.

This implies that techniques that rely on preserving plans in dynamic environments are somewhat contrary to the purpose of using cost-based optimization. Cost-based optimization attempts to produce execution plans based on statistics that accurately reflect the state of the data. Thus, you must balance the need to control plan stability with the benefit obtained from the optimizer's ability to adjust to changes in data characteristics.

How Outlines Use Hints

An outline consists primarily of a set of hints that is equivalent to the optimizer's results for the execution plan generation of a particular SQL statement. When Oracle creates an outline, Plan Stability examines the optimization results using the same data used to generate the execution plan. That is, Oracle uses the input to the execution plan to generate an outline and not the execution plan itself.

> **Note:** You cannot modify an outline. You can embed hints in SQL statements, but this has no effect on how Oracle uses outlines because Oracle considers a SQL statement that you revised with hints to be different from the original SQL statement stored in the outline.

Matching SQL Statements with Outlines

Oracle uses one of two scenarios when compiling SQL statements and matching them with outlines. The first scenario is that if you disable outline use by setting the system/session parameter USE_STORED_OUTLINES to FALSE, Oracle does not attempt to match SQL text to outlines. The second scenario involves the following two "matching" steps.

First, if you specify that Oracle must use a particular outline category, only outlines in that category are candidates for matching. Second, if the SQL text of the incoming statement exactly matches the SQL text in an outline in that category, Oracle considers both texts identical and Oracle uses the outline. Oracle considers any differences a mismatch.

Differences include spacing changes, carriage return variations, embedded hints, or even differences in comment text. These rules are identical to the rules for cursor matching.

How Oracle Stores Outlines

Oracle stores outline data in the OL$ table and hint data in the OL$HINTS table. Unless you remove them, Oracle retains outlines indefinitely. Oracle retains execution plans in cache and only recreates them if they become invalid or if the cache is not large enough to hold all of them.

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 Oracle does not use an execution plan compiled under one category to execute a SQL statement that Oracle should compile under a different category.

Parameter Settings to Enable Plan Stability

Settings for several parameters, especially those ending with the suffix "_ENABLED", must be consistent across execution environments for outlines to function properly. These parameters are:

- QUERY_REWRITE_ENABLED
- STAR_TRANSFORMATION_ENABLED
- OPTIMIZER_FEATURES_ENABLE

Creating Outlines

Oracle can automatically create outlines for all SQL statements, or you can create them for specific SQL statements. In either case, the outlines derive their input from the rule-based or cost-based optimizers.

Oracle creates stored outlines automatically when you set the parameter CREATE_STORED_OUTLINES to TRUE. When activated, Oracle creates outlines for all executed SQL statements. You can also create stored outlines for specific statements using the CREATE OUTLINE statement.

> **See Also:** For more information on the CREATE OUTLINE statement, please refer to the *Oracle8i SQL Reference*. For information on moving from the rule-based optimizer to the cost-based optimizer, refer to "Using Outlines to Move to the Cost-based Optimizer" on page 7-32.

Creating and Assigning Categories to Stored Outlines

You can create outline category names and assign outlines to them. As mentioned, this simplifies outline management because you can manipulate all outlines within a category at one time.

Both the CREATE_STORED_OUTLINES parameter and the CREATE OUTLINE statement accept category names. For either of these, if you specify a category name, Oracle assigns all subsequently created outlines to that category until you reset the category name or suspend outline generation by setting the CREATE_STORED_OUTLINES parameter to FALSE.

If you set CREATE_STORED_OUTLINES to TRUE or use the CREATE OUTLINE statement without a category name, Oracle assigns outlines to the category name of DEFAULT.

> **See Also:** For more information on the CREATE_- and USE_STORED_OUTLINES parameters, please refer to the *Oracle8i Reference*.

Using Stored Outlines

To use stored outlines when Oracle compiles a SQL statement, set the system parameter USE_STORED_OUTLINES to TRUE or to a category name. If you set USE_STORED_OUTLINES to TRUE, Oracle uses outlines in the DEFAULT category. If you specify a category with the USE_STORED_OUTLINES parameter, Oracle uses outlines in that category until you re-set the USE_STORED_OUTLINES 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, Oracle searches for an outline in the DEFAULT category.

The designated outlines only control the compilation of SQL statements that have outlines. If you set USE_STORED_OUTLINES to FALSE, Oracle does not use outlines. When you set USE_STORED_OUTLINES to FALSE and you set CREATE_STORED_OUTLINES to TRUE, Oracle creates outlines but does not use them.

When you activate the use of stored outlines, Oracle always uses the cost-based optimizer. This is because outlines rely on hints, and to be effective, most hints require the cost-based optimizer.

Viewing Outline Data

You can access information about outlines and related hint data that Oracle stores in the data dictionary from these views:

- USER_OUTLINES
- USER_OUTLINE_HINTS

For example, use this 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", for example, use this syntax:

```
SELECT HINT FROM USER_OUTLINE_HINTS WHERE NAME=’name1’;
```

See Also: If necessary, you can use the procedure to move outline tables from one tablespace to another as described under the heading "Procedure for Moving Outline Tables from One Tablespace to Another" on page 7-31.

Managing Stored Outlines with the OUTLN_PKG Package

Use procedures in the OUTLN_PKG package to manage stored outlines and their outline categories. OUTLN_PKG provides these procedures:

Table 7–2 Outline Management Procedures in the OUTLN_PKG

Procedure	Description
DROP_UNUSED	Drops outlines that Oracle has not used since they were created.
DROP_BY_CAT	Drops outlines assigned to the specified category name.
UPDATE_BY_CAT	Reassigns outlines from one category to another.

Dropping Unused Outlines

You can remove unneeded outlines using the DROP_UNUSED procedure of OUTLN_PKG. This procedure improves performance if your tablespace becomes saturated with an excessive number of outlines that Oracle will never use.

Syntax

The syntax for the DROP_UNUSED procedure is:

```
OUTLN_PKG.DROP_UNUSED;
```

Dropping Outlines within a Category

Execute the DROP_BY_CAT procedure to drop outlines within a specific category.

Syntax

The syntax and parameter for the DROP_BY_CAT procedure are:

```
OUTLN_PKG.DROP_BY_CAT(
  category_name);
```

category_name	Name of the category you want to drop.

Reassign Outlines to a Different Category

Reassign outlines from one category to another by executing the UPDATE_BY_CAT procedure.

Syntax

The syntax and parameters for the UPDATE_BY_CAT procedure are:

```
OUTLN_PKG.UPDATE_BY_CAT(
  old_category_name,
  new_category_name);
```

old_category_name	Specifies the name of the outline category that you want to reassign to a new category.
new_category_name	Specifies the name of the category to which you want to assign the outline.

Note: Use the DDL statements CREATE, DROP, and ALTER to manipulate a specific outline. For example, use the ALTER OUTLINE statement to rename an outline or change its category.

See Also: For more information about the CREATE, DROP and ALTER statements, please refer to the *Oracle8i SQL Reference*.

Moving Outline Tables

Oracle creates the USER_OUTLINES and USER_OUTLINE_HINTS views based on data in the OL$ and OL$HINTS tables respectively. Oracle creates these tables in the SYS tablespace using a schema called "OUTLN". If the outlines use too much space in the SYS tablespace, you can move them. To do this, create a separate tablespace and move the outline tables into it using the following procedure.

Procedure for Moving Outline Tables from One Tablespace to Another

Use this procedure to move outline tables:

1. Export the OL$ and OL$HINTS tables with this syntax:

   ```
   EXP OUTLN/OUTLN FILE = exp_file TABLES = 'OL$' 'OL$HINTS' SILENT=y
   ```

2. Remove the previous OL$ and OL$HINTS tables with this syntax:

   ```
   CONNECT OUTLN/outln_password;
   ```

```
DROP TABLE OL$;
CONNECT OUTLN/outln_password;
DROP TABLE OL$HINTS;
```

3. Create a new tablespace for the tables using this syntax:

```
CREATE TABLESPACE outln_ts
DATAFILE 'tspace.dat' SIZE 2MB
DEFAULT STORAGE (INITIAL 10KB NEXT 20KB
MINEXTENTS 1 MAXEXTENTS 999 PCTINCREASE 10) ONLINE;
```

4. Import the OL$ and OL$HINTS tables using this syntax:

```
IMPORT OUTLN/outln_password
FILE=exp_file TABLES = 'OL$' 'OL$HINTS' IGNORE=y SILENT=y
```

The IMPORT statement re-creates the OL$ and OL$HINTS tables in the schema named OUTLN, but the schema now resides in a new tablespace called "OUTLN_TS".

Plan Stability Procedures for the Cost-based Optimizer

This section describes procedures you can use to significantly improve performance by taking advantage of cost-based optimizer functionality. Plan Stability provides a way to preserve your system's targeted execution plans for which performance is satisfactory while also taking advantage of new cost-based optimizer features for the rest of your SQL statements.

Topics covered in this section are:

- Using Outlines to Move to the Cost-based Optimizer
- RDBMS Upgrades and the Cost-based Optimizer

Using Outlines to Move to the Cost-based Optimizer

If your application was developed using the rule-based optimizer, a considerable amount of effort may have gone into manually tuning the SQL statements to optimize performance. You can use Plan Stability to leverage the effort that has already gone into performance tuning by preserving the behavior of the application when upgrading from rule-based to cost-based optimization. By creating outlines for an application before switching to cost-based optimization, the plans generated by the rule-based optimizer can be used while statements generated by newly written applications developed after the switch will use normal, cost-based plans. To create and use outlines for an application, use the following procedure.

Note: *Carefully read this procedure and consider its implications before executing it!*

---

1. Execute syntax similar to the following to designate, for example, the RBOCAT outline category

   ```
   ALTER SESSION SET CREATE_STORED_OUTLINES = rbocat;
   ```

2. Run your application long enough to capture stored outlines for all important SQL statements.
3. Suspend outline generation with the syntax:

   ```
   ALTER SESSION SET CREATE_STORED_OUTLINES = FALSE;
   ```

4. Gather statistics with the DBMS_STATS package.
5. Alter the parameter OPTIMIZER_MODE to CHOOSE.
6. Enter this syntax to make Oracle use the outlines in category RBOCAT:

   ```
   ALTER SESSION SET USE_STORED_OUTLINES = rbocat;
   ```

7. 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, you can capture the old behavior of the query even after switching to cost-based optimization. To do this, change the optimizer mode to RULE, create an outline for the query, and then change the optimizer mode back to CHOOSE.

RDBMS Upgrades and the Cost-based Optimizer

When upgrading to a new version of Oracle under cost-based optimization, there is always a possibility that some SQL statements will have their execution plans changed due to changes in the optimizer. While such changes benefit performance in the vast majority of cases, you might have some applications that perform well and where you would consider any changes in their behavior to be an unnecessary risk. For such applications, you can create outlines before the upgrade using the following procedure.

Note: *Carefully read this procedure and consider its implications before executing it!*

1. Enter this syntax to enable outline creation:

   ```
   ALTER SESSION SET CREATE_STORED_OUTLINES = ALL_QUERIES;
   ```

2. Run the application long enough to capture stored outlines for all critical SQL statements.
3. Enter this syntax to suspend outline generation:

   ```
   ALTER SESSION SET CREATE_STORED_OUTLINES = FALSE;
   ```

4. Upgrade the production system to the new version of the RDBMS.
5. Run the application.

After the upgrade, you can enable the use of stored outlines, or alternatively, you can use the outlines that were stored as a backup if you find that some statements exhibit performance degradation after the upgrade.

With the latter approach, you can selectively use the stored outlines for such problematic statements as follows:

1. For each problematic SQL statement, change the CATEGORY of the associated stored outline to a category name similar to this:

   ```
   ALTER OUTLINE outline_name CHANGE CATEGORY TO problemcat;
   ```

2. Enter this syntax to make Oracle use outlines from the category "PROBLEMCAT".

   ```
   ALTER SESSION SET USE_STORED_OUTLINES = problemcat;
   ```

Upgrading with a Test System

A test system, separate from the production system, can be useful for conducting experiments with optimizer behavior in conjunction with an upgrade. You can migrate statistics from the production system to the test system using import/export. This may alleviate the need to fill the tables in the test system with data.

You can move outlines between the systems by category. For example, once you create outlines in the PROBLEMCAT category, export them by category using the query-based export option. This is a convenient and efficient way to export only selected outlines from one database to another without exporting all outlines in the source database. To do this, issue these statements:

```
EXP OUTLN/outln_password FILE=<exp-file> TABLES= ’OL$’ ’OL$HINTS’
QUERY=’WHERE CATEGORY="problemcat"'
```

Using Hints

As an application designer, you may know information about your data that the optimizer does not know. For example, you may know that a certain index is more selective for certain queries. Based on this information, you may be able to choose a more efficient execution plan than the optimizer. In such a case, use hints to force the optimizer to use the optimal execution plan.

Hints allow you to make decisions usually made by the optimizer. You can use hints to specify:

- The optimization approach for a SQL statement
- The goal of the cost-based approach for a SQL statement
- The access path for a table accessed by the statement
- The join order for a join statement
- A join operation in a join statement

Note: The use of hints involves extra code that must also be managed, checked, and controlled.

Specifying Hints

Hints apply only to the optimization of the statement block in which they appear. A statement block is any one of the following statements or parts of statements:

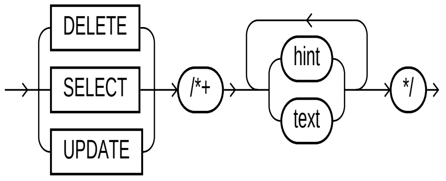
- A simple SELECT, UPDATE, or DELETE statement
- A parent statement or subquery of a complex statement
- A part of a compound query

For example, a compound query consisting of two component queries combined by the UNION operator has two statement blocks, one for each component query. For this reason, hints in the first component query apply only to its optimization, not to the optimization of the second component query.

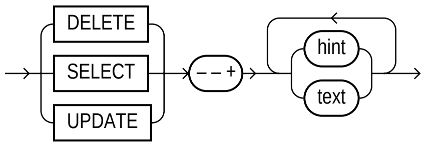
You can send hints for a SQL statement to the optimizer by enclosing them in a comment within the statement.

See Also: For more information on comments, see *Oracle8i SQL Reference.*

A statement block can have only one comment containing hints. This comment can only follow the SELECT, UPDATE, or DELETE keyword. The syntax diagrams show the syntax for hints contained in both styles of comments that Oracle supports within a statement block.



or:



where:

DELETE
SELECT
UPDATE
	Is a DELETE, SELECT, or UPDATE keyword that begins a statement block. Comments containing hints can appear only after these keywords.

+	Is a plus sign that causes Oracle to interpret the comment as a list of hints. The plus sign must immediately follow the comment delimiter (no space is permitted).
hint	Is one of the hints discussed in this section. If the comment contains multiple hints, each pair of hints must be separated by at least one space.
text	Is other commenting text that can be interspersed with the hints.

If you specify hints incorrectly, Oracle ignores them but does not return an error:

- Oracle ignores hints if the comment containing them does not follow a DELETE, SELECT, or UPDATE keyword.
- Oracle ignores hints containing syntax errors, but considers other correctly specified hints within the same comment.

- Oracle ignores combinations of conflicting hints, but considers other hints within the same comment.
- Oracle ignores hints in all SQL statements in those environments that use PL/SQL Version 1, such as SQL*Forms Version 3 triggers, Oracle Forms 4.5, and Oracle Reports 2.5.

Other conditions specific to index type appear later in this chapter.

The optimizer recognizes hints only when using the cost-based approach. If you include a hint (except the RULE hint) in a statement block, the optimizer automatically uses the cost-based approach.

The following sections show the syntax of each hint.

Hints for Optimization Approaches and Goals

The hints described in this section allow you to choose between the cost-based and the rule-based optimization approaches and, with the cost-based approach, either a goal of best throughput or best response time.

- ALL_ROWS
- FIRST_ROWS
- CHOOSE
- RULE

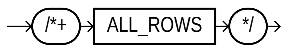
If a SQL statement has a hint specifying an optimization approach and goal, the optimizer uses the specified approach regardless of the presence or absence of statistics, the value of the OPTIMIZER_MODE initialization parameter, and the OPTIMIZER_MODE parameter of the ALTER SESSION statement.

Note: The optimizer goal applies only to queries submitted directly. Use hints to determine the access path for any SQL statements submitted from within PL/SQL. The ALTER SESSION... SET OPTIMIZER_MODE statement does not affect SQL that is run from within PL/SQL.

ALL_ROWS

The ALL_ROWS hint explicitly chooses the cost-based approach to optimize a statement block with a goal of best throughput (that is, minimum total resource consumption).

The syntax of this hint is as follows:



For example, the optimizer uses the cost-based approach to optimize this statement for best throughput:

```
SELECT /*+ ALL_ROWS */ empno, ename, sal, job
  FROM emp
 WHERE empno = 7566;
```

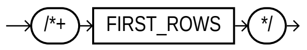
FIRST_ROWS

The FIRST_ROWS hint explicitly chooses the cost-based approach to optimize a statement block with a goal of best response time (minimum resource usage to return first row).

This hint causes the optimizer to make these choices:

- If an index scan is available, the optimizer may choose it over a full table scan.
- If an index scan is available, the optimizer may choose a nested loops join over a sort-merge join whenever the associated table is the potential inner table of the nested loops.
- If an index scan is made available by an ORDER BY clause, the optimizer may choose it to avoid a sort operation.

The syntax of this hint is as follows:



For example, the optimizer uses the cost-based approach to optimize this statement for best response time:

```
SELECT /*+ FIRST_ROWS */ empno, ename, sal, job
  FROM emp
 WHERE empno = 7566;
```

The optimizer ignores this hint in DELETE and UPDATE statement blocks and in SELECT statement blocks that contain any of the following syntax:

- Set operators (UNION, INTERSECT, MINUS, UNION ALL)

- GROUP BY clause
- FOR UPDATE clause
- Aggregate functions
- DISTINCT operator

These statements cannot be optimized for best response time because Oracle must retrieve all rows accessed by the statement before returning the first row. If you specify this hint in any of these statements, the optimizer uses the cost-based approach and optimizes for best throughput.

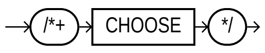
If you specify either the ALL_ROWS or FIRST_ROWS hint in a SQL statement and the data dictionary does not have statistics about tables accessed by the statement, the optimizer uses default statistical values (such as allocated storage for such tables) to estimate the missing statistics and subsequently to choose an execution plan.

These estimates may not be as accurate as those generated by the ANALYZE statement. Therefore, use the ANALYZE statement to generate statistics for all tables accessed by statements that use cost-based optimization. If you specify hints for access paths or join operations along with either the ALL_ROWS or FIRST_ROWS hint, the optimizer gives precedence to the access paths and join operations specified by the hints.

CHOOSE

The CHOOSE hint causes the optimizer to choose between the rule-based and cost-based approaches for a SQL statement. The optimizer bases its selection on the presence of statistics for the tables accessed by the statement. If the data dictionary has statistics for at least one of these tables, the optimizer uses the cost-based approach and optimizes with the goal of best throughput. If the data dictionary does not have statistics for these tables, it uses the rule-based approach.

The syntax of this hint is as follows:



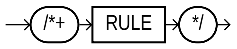
For example:

```
SELECT /*+ CHOOSE */ empno, ename, sal, job
  FROM emp
 WHERE empno = 7566;
```

RULE

The RULE hint explicitly chooses rule-based optimization for a statement block. It also makes the optimizer ignore other hints specified for the statement block.

The syntax of this hint is as follows:



For example, the optimizer uses the rule-based approach for this statement:

```
SELECT --+ RULE
  empno, ename, sal, job
  FROM emp
  WHERE empno = 7566;
```

The RULE hint, along with the rule-based approach, may not be supported in future versions of Oracle.

Hints for Access Methods

Each hint described in this section suggests an access method for a table.

- FULL
- ROWID
- CLUSTER
- HASH
- HASH_AJ
- HASH_SJ
- INDEX
- INDEX_ASC
- INDEX_COMBINE
- INDEX_JOIN
- INDEX_DESC
- INDEX_FFS
- NO_INDEX

- MERGE_AJ
- MERGE_SJ
- AND_EQUAL
- USE_CONCAT
- NO_EXPAND
- REWRITE
- NOREWRITE

Specifying one of these hints causes the optimizer to choose the specified access path only if the access path is available based on the existence of an index or cluster and on the syntactic constructs of the SQL statement. If a hint specifies an unavailable access path, the optimizer ignores it.

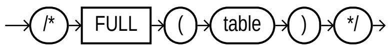
You must specify the table to be accessed exactly as it appears in the statement. If the statement uses an alias for the table, use the alias rather than the table name in the hint. The table name within the hint should not include the schema name if the schema name is present in the statement.

Note: For access path hints, Oracle ignores the hint if you specify the SAMPLE option in the FROM clause of a SELECT statement. For more information on the SAMPLE option, please refer to *Oracle8i Concepts* and *Oracle8i Reference*.

FULL

The FULL hint explicitly chooses a full table scan for the specified table.

The syntax of this hint is as follows:



where *table* specifies the name or alias of the table on which the full table scan is to be performed.

For example, Oracle performs a full table scan on the ACCOUNTS table to execute this statement, even if there is an index on the ACCNO column that is made available by the condition in the WHERE clause:

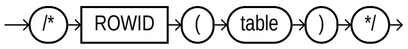
```
SELECT /*+ FULL(A) DON'T USE THE INDEX ON ACCNO */ ACCNO, BAL
```

```
 FROM ACCOUNTS A
WHERE ACCNO = 7086854;
```

Note: Because the ACCOUNTS table has alias "A", the hint must refer to the table by its alias rather than by its name. Also, do not specify schema names in the hint even if they are specified in the FROM clause.

ROWID

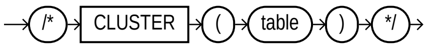
The ROWID hint explicitly chooses a table scan by rowid for the specified table. The syntax of the ROWID hint is:



where *table* specifies the name or alias of the table on which the table access by rowid is to be performed.

CLUSTER

The CLUSTER hint explicitly chooses a cluster scan to access the specified table. It applies only to clustered objects. The syntax of the CLUSTER hint is:



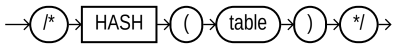
where *table* specifies the name or alias of the table to be accessed by a cluster scan.

The following example illustrates the use of the CLUSTER hint.

```
SELECT --+ CLUSTER
  EMP ENAME, DEPTNO
  FROM EMP, DEPT
 WHERE DEPTNO = 10 AND
    EMP.DEPTNO = DEPT.DEPTNO;
```

HASH

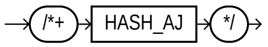
The HASH hint explicitly chooses a hash scan to access the specified table. It applies only to tables stored in a cluster. The syntax of the HASH hint is:



where *table* specifies the name or alias of the table to be accessed by a hash scan.

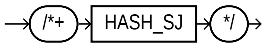
HASH_AJ

The HASH_AJ hint transforms a NOT IN subquery into a hash anti-join to access the specified table. The syntax of the HASH_AJ hint is:



HASH_SJ

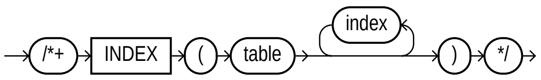
The HASH_SJ hint transforms a correlated EXISTS subquery into a hash semi-join to access the specified table. The syntax of the HASH_SJ hint is:



INDEX

The INDEX hint explicitly chooses an index scan for the specified table. You can use the INDEX hint for domain, B*-tree, and bitmap indexes. However, Oracle recommends using INDEX_COMBINE rather than INDEX for bitmap indexes because it is a more versatile hint.

The syntax of the INDEX hint is:



where:

table	Specifies the name or alias of the table associated with the index to be scanned.

index	Specifies an index on which an index scan is to be performed.

This hint may optionally specify one or more indexes:

- If this hint specifies a single available index, the optimizer performs a scan on this index. The optimizer does not consider a full table scan or a scan on another index on the table.
- If this hint specifies a list of available indexes, the optimizer considers the cost of a scan on each index in the list and then performs the index scan with the lowest cost. The optimizer may also choose to scan multiple indexes from this list and merge the results, if such an access path has the lowest cost. The optimizer does not consider a full table scan or a scan on an index not listed in the hint.
- If this hint specifies no indexes, the optimizer considers the cost of a scan on each available index on the table and then performs the index scan with the lowest cost. The optimizer may also choose to scan multiple indexes and merge the results, if such an access path has the lowest cost. The optimizer does not consider a full table scan.

For example, consider this query that selects the name, height, and weight of all male patients in a hospital:

```
SELECT name, height, weight
  FROM patients
 WHERE sex = 'm';
```

Assume there is an index on the SEX column and that this column contains the values M and F. If there are equal numbers of male and female patients in the hospital, the query returns a relatively large percentage of the table's rows and a full table scan is likely to be faster than an index scan. However, if a very small percentage of the hospital's patients are male, the query returns a relatively small percentage of the table's rows and an index scan is likely to be faster than a full table scan.

The number of occurrences of each distinct column value is not available to the optimizer. The cost-based approach assumes that each value has an equal probability of appearing in each row. For a column having only two distinct values, the optimizer assumes each value appears in 50% of the rows, so the cost-based approach is likely to choose a full table scan rather than an index scan.

If you know that the value in the WHERE clause of your query appears in a very small percentage of the rows, you can use the INDEX hint to force the optimizer to choose an index scan. In this statement, the INDEX hint explicitly chooses an index scan on the SEX_INDEX, the index on the SEX column:

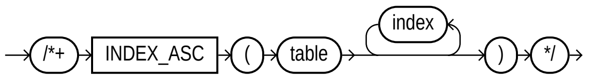
```
SELECT /*+ INDEX(PATIENTS SEX_INDEX) USE SEX_INDEX, SINCE THERE ARE FEW
             MALE PATIENTS  */
       NAME, HEIGHT, WEIGHT
  FROM PATIENTS
 WHERE SEX = 'M';
```

The INDEX hint applies to inlist predicates; it forces the optimizer to use the hinted index, if possible, for an inlist predicate. Multi-column inlists will not use an index.

This hint is useful if you are using distributed query optimization. For more information about this, please refer to *Oracle8i Distributed Database Systems*.

INDEX_ASC

The INDEX_ASC hint explicitly chooses an index scan for the specified table. If the statement uses an index range scan, Oracle scans the index entries in ascending order of their indexed values. The syntax of the INDEX_ASC hint is:



Each parameter serves the same purpose as in the INDEX hint.

Because Oracle's default behavior for a range scan is to scan index entries in ascending order of their indexed values, this hint does not specify anything more than the INDEX hint. However, you may want to use the INDEX_ASC hint to specify ascending range scans explicitly, should the default behavior change.

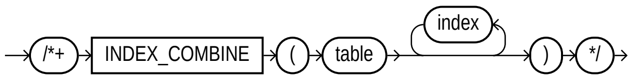
This hint is useful if you are using distributed query optimization.

> **See Also:** For more information about the INDEX_ASC hint, please refer to *Oracle8i Distributed Database Systems*.

INDEX_COMBINE

The INDEX_COMBINE hint explicitly chooses a bitmap access path for the table. If no indexes are given as arguments for the INDEX_COMBINE hint, the optimizer uses whatever Boolean combination of bitmap indexes has the best cost estimate for the table. If certain indexes are given as arguments, the optimizer tries to use some

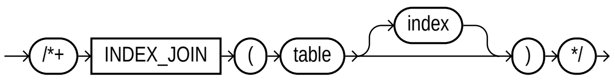
Boolean combination of those particular bitmap indexes. The syntax of INDEX_COMBINE is:



This hint is useful for bitmap indexes and if you are using distributed query optimization. For more information about this, please refer to *Oracle8i Distributed Database Systems.*

INDEX_JOIN

The INDEX_JOIN hint explicitly instructs the optimizer to use an index join as an access path. For the hint to have a positive effect, a sufficiently small number of indexes must exist that contain all the columns required to resolve the query.

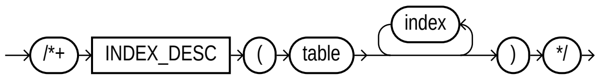


where:

table	Specifies the name or alias of the table associated with the index to be scanned.
index	Specifies an index on which an index scan is to be performed.

INDEX_DESC

The INDEX_DESC hint explicitly chooses an index scan for the specified table. If the statement uses an index range scan, Oracle scans the index entries in descending order of their indexed values. The syntax of the INDEX_DESC hint is:

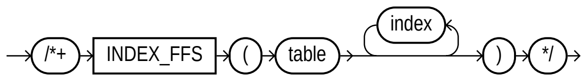


Each parameter serves the same purpose as in the INDEX hint. This hint has no effect on SQL statements that access more than one table. Such statements always perform range scans in ascending order of the indexed values.

This hint is useful if you are using distributed query optimization. For more information about this, please refer to *Oracle8i Distributed Database Systems*.

INDEX_FFS

This hint causes a fast full index scan to be performed rather than a full table scan. The syntax of INDEX_FFS is:

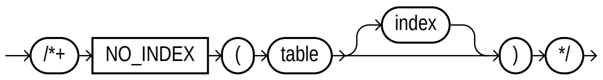


See Also: "Using Fast Full Index Scans" on page 6-8.

This hint is useful if you are using distributed query optimization. For more information about this, please refer to *Oracle8i Distributed Database Systems*.

NO_INDEX

The NO_INDEX hint explicitly disallows a set of indexes for the specified table. The syntax of the NO_INDEX hint is:



Use this hint to optionally specify one or more indexes:

- If this hint specifies a single available index, the optimizer does not consider a scan on this index. Other indexes not specified are still considered.
- If this hint specifies a list of available indexes, the optimizer does not consider a scan on any of the specified indexes. Other indexes not specified in the list are still considered.
- If this hint specifies no indexes, the optimizer does not consider a scan on any index on the table. This behavior is the same as a NO_INDEX hint that specifies a list of all available indexes for the table.

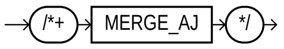
The NO_INDEX hint applies to function-based, B*-tree, bitmap, cluster, or domain indexes.

If a NO_INDEX hint and an index hint (INDEX, INDEX_ASC, INDEX_DESC, INDEX_COMBINE, or INDEX_FFS) both specify the same indexes, then both the NO_INDEX hint and the index hint are ignored for the specified indexes and the optimizer will consider the specified indexes.

This hint is useful if you are using distributed query optimization. For more information about this, please refer to *Oracle8i Distributed Database Systems*.

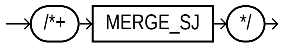
MERGE_AJ

The MERGE_AJ hint transforms a NOT IN subquery into a merge anti-join to access the specified table. The syntax of the MERGE_AJ hint is:



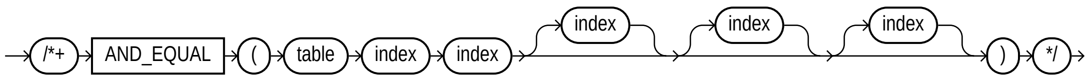
MERGE_SJ

The MERGE_SJ hint transforms a correlated EXISTS subquery into a merge semi-join to access the specified table. The syntax of the MERGE_SJ hint is:



AND_EQUAL

The AND_EQUAL hint explicitly chooses an execution plan that uses an access path that merges the scans on several single-column indexes. The syntax of the AND_EQUAL hint is:



where:

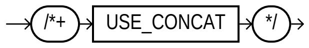
table	Specifies the name or alias of the table associated with the indexes to be merged.
index	Specifies an index on which an index scan is to be performed. You must specify at least two indexes. You cannot specify more than five.

USE_CONCAT

The USE_CONCAT hint forces combined OR conditions in the WHERE clause of a query to be transformed into a compound query using the UNION ALL set operator. Normally, this transformation occurs only if the cost of the query using the concatenations is cheaper than the cost without them.

The USE_CONCAT hint turns off inlist processing and OR-expands all disjunctions, including inlists.

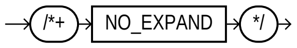
The syntax of this hint is:



NO_EXPAND

The NO_EXPAND hint prevents the cost-based optimizer from considering OR-expansion for queries having OR conditions or INLISTS in the WHERE clause. Normally, the optimizer would consider using OR expansion and use this method if it decides the cost is lower than not using it.

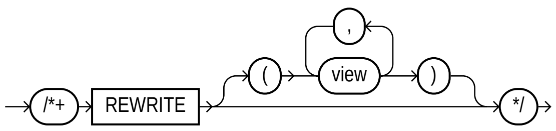
The syntax of this hint is:



REWRITE

Use the REWRITE hint with or without a view list. If you use REWRITE with a view list and the list contains an eligible materialized view, Oracle uses that view regardless of its cost. Oracle does not consider views outside of the list. If you do not specify a view list, Oracle searches for an eligible materialized view and always uses it regardless of its cost.

The syntax of this hint is:

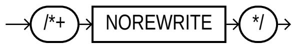


See Also: For more information on materialized views, please refer to *Oracle8i Concepts* and to *Oracle8i Application Developer's Guide - Fundamentals.*

NOREWRITE

Use the NOREWRITE hint on any query block of a request. This hint disables query rewrite for the query block, overriding the setting of the parameter QUERY_REWRITE_ENABLED.

The syntax of this hint is:



Hints for Join Orders

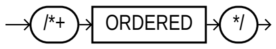
The hints in this section suggest join orders:

- ORDERED
- STAR

ORDERED

The ORDERED hint causes Oracle to join tables in the order in which they appear in the FROM clause.

The syntax of this hint is:



For example, this statement joins table TAB1 to table TAB2 and then joins the result to table TAB3:

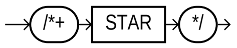
```
SELECT /*+ ORDERED */ TAB1.COL1, TAB2.COL2, TAB3.COL3
  FROM TAB1, TAB2, TAB3
 WHERE TAB1.COL1 = TAB2.COL1
      AND TAB2.COL1 = TAB3.COL1;
```

If you omit the ORDERED hint from a SQL statement performing a join, the optimizer chooses the order in which to join the tables. You may want to use the ORDERED hint to specify a join order if you know something about the number of rows selected from each table that the optimizer does not. Such information would allow you to choose an inner and outer table better than the optimizer could.

STAR

The STAR hint forces a star query plan to be used if possible. A star plan has the largest table in the query last in the join order and joins it with a nested loops join on a concatenated index. The STAR hint applies when there are at least 3 tables, the large table's concatenated index has at least 3 columns, and there are no conflicting access or join method hints. The optimizer also considers different permutations of the small tables.

The syntax of this hint is:



Usually, if you analyze the tables, the optimizer selects an efficient star plan. You can also use hints to improve the plan. The most precise method is to order the tables in the FROM clause in the order of the keys in the index, with the large table last. Then use the following hints:

```
/*+ ORDERED USE_NL(FACTS) INDEX(FACTS FACT_CONCAT) */
```

Where "facts" is the table and "fact_concat" is the index. A more general method is to use the STAR hint.

> **See Also:** *Oracle8i Concepts* for more information about star plans.

Hints for Join Operations

Each hint described in this section suggests a join operation for a table.

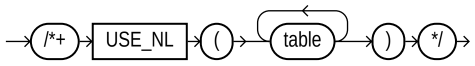
- USE_NL
- USE_MERGE
- USE_HASH
- DRIVING_SITE
- HASH_AJ and MERGE_AJ
- HASH_SJ and MERGE_SJ

You must specify a table to be joined exactly as it appears in the statement. If the statement uses an alias for the table, you must use the alias rather than the table name in the hint. The table name within the hint should not include the schema name, if the schema name is present in the statement.

Use of the USE_NL and USE_MERGE hints is recommended with the ORDERED hint. Oracle uses these hints when the referenced table is forced to be the inner table of a join, and they are ignored if the referenced table is the outer table.

USE_NL

The USE_NL hint causes Oracle to join each specified table to another row source with a nested loops join using the specified table as the inner table. The syntax of the USE_NL hint is:



where *table* is the name or alias of a table to be used as the inner table of a nested loops join.

For example, consider this statement, which joins the ACCOUNTS and CUSTOMERS tables. Assume that these tables are not stored together in a cluster:

```
SELECT ACCOUNTS.BALANCE, CUSTOMERS.LAST_NAME, CUSTOMERS.FIRST_NAME
  FROM ACCOUNTS, CUSTOMERS
 WHERE ACCOUNTS.CUSTNO = CUSTOMERS.CUSTNO;
```

Since the default goal of the cost-based approach is best throughput, the optimizer will choose either a nested loops operation or a sort-merge operation to join these tables, depending on which is likely to return all the rows selected by the query more quickly.

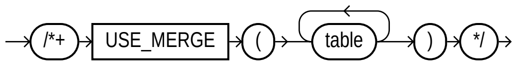
However, you may want to optimize the statement for best response time, or the minimal elapsed time necessary to return the first row selected by the query, rather than best throughput. If so, you can force the optimizer to choose a nested loops join by using the USE_NL hint. In this statement, the USE_NL hint explicitly chooses a nested loops join with the CUSTOMERS table as the inner table:

```
SELECT /*+ ORDERED USE_NL(CUSTOMERS) USE N-L TO GET FIRST ROW FASTER */
ACCOUNTS.BALANCE, CUSTOMERS.LAST_NAME, CUSTOMERS.FIRST_NAME
  FROM ACCOUNTS, CUSTOMERS
 WHERE ACCOUNTS.CUSTNO = CUSTOMERS.CUSTNO;
```

In many cases, a nested loops join returns the first row faster than a sort-merge join. A nested loops join can return the first row after reading the first selected row from one table and the first matching row from the other and combining them, while a sort-merge join cannot return the first row until after reading and sorting all selected rows of both tables and then combining the first rows of each sorted row source.

USE_MERGE

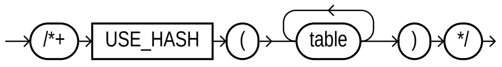
The USE_MERGE hint causes Oracle to join each specified table with another row source with a sort-merge join. The syntax of the USE_MERGE hint is:



where *table* is a table to be joined to the row source resulting from joining the previous tables in the join order using a sort-merge join.

USE_HASH

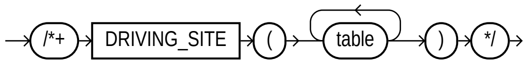
The USE_HASH hint causes Oracle to join each specified table with another row source with a hash join. The syntax of the USE_HASH hint is:



where *table* is a table to be joined to the row source resulting from joining the previous tables in the join order using a hash join.

DRIVING_SITE

The DRIVING_SITE hint forces query execution to be done at a different site than that selected by Oracle. This hint can be used with either rule-based or cost-based optimization. The syntax of this hint is:



where *table* is the name or alias for the table at which site the execution should take place.

Example:

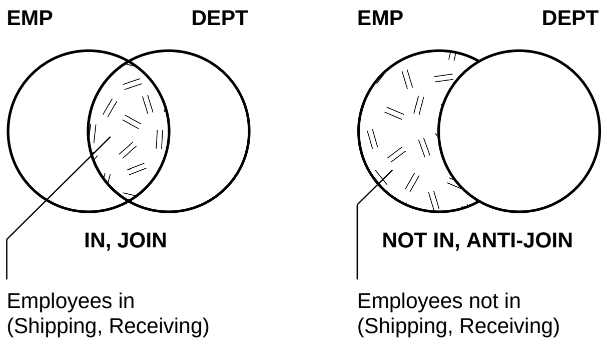
```
SELECT /*+DRIVING_SITE(DEPT)*/ * FROM EMP, DEPT@RSITE
 WHERE EMP.DEPTNO = DEPT.DEPTNO;
```

If this query is executed without the hint, rows from DEPT will be sent to the local site and the join will be executed there. With the hint, the rows from EMP will be sent to the remote site and the query will be executed there, returning the result to the local site.

This hint is useful if you are using distributed query optimization. For more information about this, please refer to *Oracle8i Distributed Database Systems.*

HASH_AJ and MERGE_AJ

Figure 7–1 Parallel Hash Anti-join


EMP
DEPT
EMP
DEPT
IN, JOIN
NOT IN, ANTI-JOIN
Employees in (Shipping, Receiving)
Employees not in (Shipping, Receiving)

As illustrated in Figure 7–1, the SQL IN predicate can be evaluated using a join to intersect two sets. Thus emp.deptno can be joined to dept.deptno to yield a list of employees in a set of departments.

Alternatively, the SQL NOT IN predicate can be evaluated using an anti-join to subtract two sets. Thus emp.deptno can be anti-joined to dept.deptno to select all employees who are not in a set of departments. Thus you can get a list of all employees who are not in the Shipping or Receiving departments.

For a specific query, place the MERGE_AJ or HASH_AJ hints into the NOT IN subquery. MERGE_AJ uses a sort-merge anti-join and HASH_AJ uses a hash anti-join.

For example:

```
SELECT * FROM EMP
WHERE ENAME LIKE 'J%' AND
DEPTNO IS NOT NULL AND
DEPTNO NOT IN (SELECT /*+ HASH_AJ */ DEPTNO FROM DEPT
WHERE DEPTNO IS NOT NULL AND
LOC = 'DALLAS');
```

If you wish the anti-join transformation always to occur if the conditions in the previous section are met, set the ALWAYS_ANTI_JOIN initialization parameter to MERGE or HASH. The transformation to the corresponding anti-join type then takes place whenever possible.

HASH_SJ and MERGE_SJ

For a specific query, place the HASH_SJ or MERGE_SJ hint into the EXISTS subquery. HASH_SJ uses a hash semi-join and MERGE_SJ uses a sort merge semi-join. For example:

```
SELECT * FROM T1
WHERE EXISTS (SELECT /*+ HASH_SJ */ * FROM T
WHERE T1.C1 = T2.C1
AND T2.C3 > 5);
```

This converts the subquery into a special type of join between t1 and t2 that preserves the semantics of the subquery; that is, even if there is more than one matching row in t2 for a row in t1, the row in t1 will be returned only once.

A subquery will be evaluated as a semi-join only with these limitations:

- There can only be one table in the subquery.
- The outer query block must not itself be a subquery.
- The subquery must be correlated with an equality predicate.
- The subquery must have no GROUP BY, CONNECT BY, or ROWNUM references.

If you wish the semi-join transformation always to occur if the conditions in the previous section are met, set the ALWAYS_SEMI_JOIN initialization parameter to HASH or MERGE. The transformation to the corresponding semi-join type then takes place whenever possible.

Hints for Parallel Execution

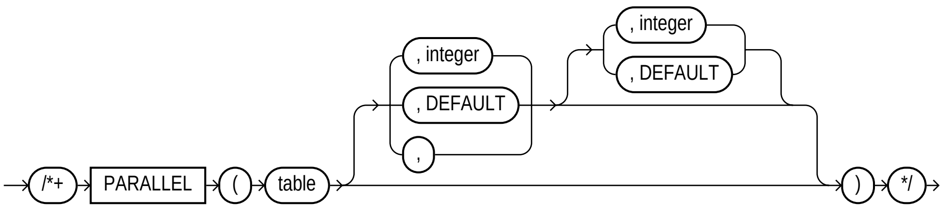
The hints described in this section determine how statements are parallelized or not parallelized when using parallel execution.

- PARALLEL
- NOPARALLEL
- PQ_DISTRIBUTE
- APPEND
- NOAPPEND
- PARALLEL_INDEX
- NOPARALLEL_INDEX

See Also: Chapter 26, "Tuning Parallel Execution".

PARALLEL

The PARALLEL hint lets you specify the desired number of concurrent servers that can be used for a parallel operation. The hint applies to the INSERT, UPDATE, and DELETE portions of a statement as well as to the table scan portion. If any parallel restrictions are violated, the hint is ignored. The syntax is:



The PARALLEL hint must use the table alias if an alias is specified in the query. The hint can then take two values separated by commas after the table name. The first value specifies the degree of parallelism for the given table, the second value specifies how the table is to be split among the instances of a parallel server. Specifying DEFAULT or no value signifies that the query coordinator should examine the settings of the initialization parameters (described in a later section) to determine the default degree of parallelism.

In the following example, the PARALLEL hint overrides the degree of parallelism specified in the EMP table definition:

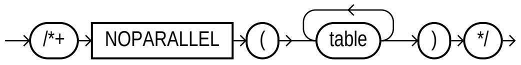
```
SELECT /*+ FULL(SCOTT_EMP) PARALLEL(SCOTT_EMP, 5) */ ENAME
  FROM SCOTT.EMP SCOTT_EMP;
```

In the next example, the PARALLEL hint overrides the degree of parallelism specified in the EMP table definition and tells the optimizer to use the default degree of parallelism determined by the initialization parameters. This hint also specifies that the table should be split among all of the available instances, with the default degree of parallelism on each instance.

```
SELECT /*+ FULL(SCOTT_EMP) PARALLEL(SCOTT_EMP, DEFAULT,DEFAULT) */ ENAME
  FROM SCOTT.EMP SCOTT_EMP;
```

NOPARALLEL

You can use the NOPARALLEL hint to override a PARALLEL specification in the table clause. In general, hints take precedence over table clauses. The syntax of this hint is:



The following example illustrates the NOPARALLEL hint:

```
SELECT /*+ NOPARALLEL(scott_emp) */ ename
  FROM scott.emp scott_emp;
```

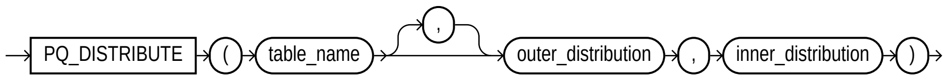
The NOPARALLEL hint is equivalent to specifying the hint.

PQ_DISTRIBUTE

Use the PQ_DISTRIBUTE hint to improve parallel join operation performance. Do this by specifying how rows of joined tables should be distributed among producer and consumer query servers. Using this hint overrides decisions the optimizer would normally make.

Use the EXPLAIN PLAN statement to identify the distribution chosen by the optimizer. The optimizer ignores the distribution hint if both tables are serial. For more information on how Oracle parallelizes join operations, please refer to *Oracle8i Concepts*.

The syntax of the distribution hint is:



where:

table_name	Is the name or alias of a table to be used as the inner table of a join.
outer_distribution	Is the distribution for the outer table.
inner_distribution	Is the distribution for the inner table.

There are six combinations for table distribution:

- Hash / Hash
- Broadcast / None
- None / Broadcast
- Partition / None
- None / Partition
- None / None

Only a subset of distribution method combinations for the joined tables is valid as explained in Table 7–3. For example:

Table 7–3 Distribution Hint Combinations

Distribution	Interpretation
Hash, Hash	Maps the rows of each table to consumer query servers using a hash function on the join keys. When mapping is complete, each query server performs the join between a pair of resulting partitions. This hint is recommended when the tables are comparable in size and the join operation is implemented by hash-join or sort-merge join.
Broadcast, None	All rows of the outer table are broadcast to each query server. The inner table rows are randomly partitioned. This hint is recommended when the outer table is very small compared to the inner table. A rule-of-thumb is: *Use the Broadcast/None hint if the size of the inner table * number of query servers < size of the outer table.*
None, Broadcast	All rows of the inner table are broadcast to each consumer query server. The outer table rows are randomly partitioned. This hint is recommended when the inner table is very small compared to the outer table. A rule-of-thumb is: *Use the None/Broadcast hint if the size of the inner table * number of query servers < size of the outer table.*
Partition, None	Maps the rows of the outer table using the partitioning of the inner table. The inner table must be partitioned on the join keys. This hint is recommended when the number of partitions of the outer table is equal to or nearly equal to a multiple of the number of query servers, for example, 14 partitions and 15 query servers. **Note:** The optimizer ignores this hint if the inner table is not partitioned or not equijoined on the partitioning key.
None, Partition	Maps the rows of the inner table using the partitioning of the outer table. The outer table must be partitioned on the join keys. This hint is recommended when the number of partitions of the outer table is equal to or nearly equal to a multiple of the number of query servers, for example, 14 partitions and 15 query servers. **Note:** The optimizer ignores this hint if the outer table is not partitioned or not equijoined on the partitioning key.

Table 7–3 Distribution Hint Combinations

Distribution	Interpretation
None, None	Each query server performs the join operation between a pair of matching partitions, one from each table. Both tables must be equi-partitioned on the join keys.

Examples: Given two tables R and S that are joined using a hash-join, the following query contains a hint to use hash distribution:

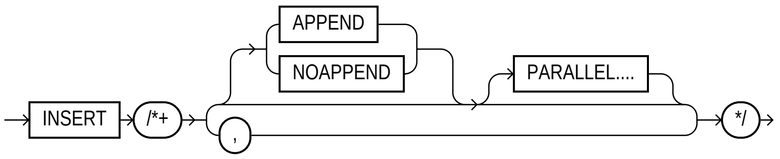
```
SELECT <column_list>
/*+ORDERED PQ_DISTRIBUTE(S HASH HASH) USE_HASH (S)*/
FROM R,S
WHERE R.C=S.C;
```

To broadcast the outer table R, the query should be:

```
SELECT <column list>
/*+ORDERED PQ_DISTRIBUTE(S BROADCAST NONE) USE_HASH (S) */
FROM R,S
WHERE R.C=S.C;
```

APPEND

When you use the APPEND hint for INSERT, data is simply appended to a table. Existing free space in the block is not used. The syntax of this hint is:



If INSERT is parallelized using the PARALLEL hint or clause, append mode will be used by default. You can use NOAPPEND to override append mode. The APPEND hint applies to both serial and parallel insert.

The append operation is performed in LOGGING or NOLOGGING mode, depending on whether the [NO]LOGGING option is set for the table in question. Use the ALTER TABLE... [NO]LOGGING statement to set the appropriate value.

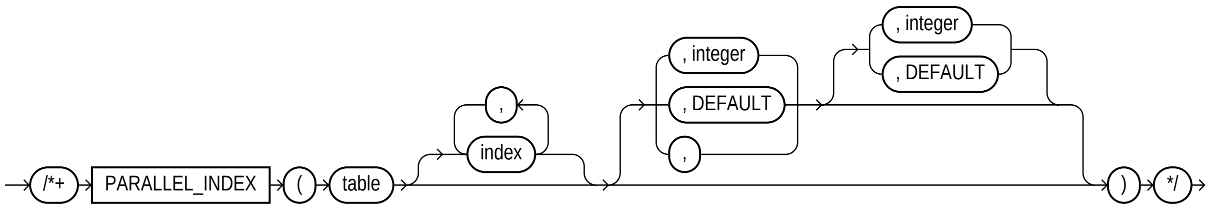
Certain restrictions apply to the APPEND hint; these are detailed in *Oracle8i Concepts*. If any of these restrictions are violated, the hint will be ignored.

NOAPPEND

Use NOAPPEND to override append mode.

PARALLEL_INDEX

Use the PARALLEL_INDEX hint to specify the desired number of concurrent servers that can be used to parallelize index range scans for partitioned indexes. The syntax of the PARALLEL_INDEX hint is:



where:

table	Specifies the name or alias of the table associated with the index to be scanned.
index	Specifies an index on which an index scan is to be performed (optional).

The hint can take two values separated by commas after the table name. The first value specifies the degree of parallelism for the given table. The second value specifies how the table is to be split among the instances of a parallel server. Specifying DEFAULT or no value signifies the query coordinator should examine the settings of the initialization parameters (described in a later section) to determine the default degree of parallelism.

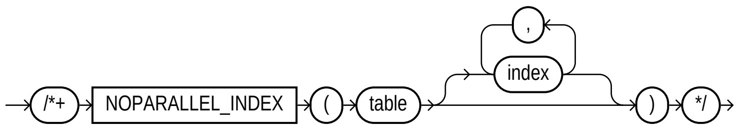
For example:

```
SELECT /*+ PARALLEL_INDEX(TABLE1,INDEX1, 3, 2) +/;
```

In this example there are 3 parallel execution processes to be used on each of 2 instances.

NOPARALLEL_INDEX

Use the NOPARALLEL_INDEX hint to override a PARALLEL attribute setting on an index. In this way you can avoid a parallel index scan operation. The syntax of this hint is:



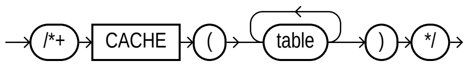
Additional Hints

Several additional hints are included in this section:

- CACHE
- NOCACHE
- MERGE
- NO_MERGE
- PUSH_JOIN_PRED
- NO_PUSH_JOIN_PRED
- PUSH_SUBQ
- STAR_TRANSFORMATION
- ORDERED_PREDICATES

CACHE

The CACHE hint specifies that the blocks retrieved for this table are placed at the most recently used end of the LRU list in the buffer cache when a full table scan is performed. This option is useful for small lookup tables. The syntax of this hint is:

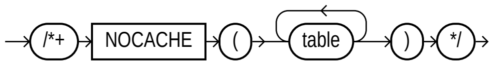


In the following example, the CACHE hint overrides the table's default caching specification:

```
SELECT /*+ FULL (SCOTT_EMP) CACHE(SCOTT_EMP) */ ENAME
  FROM SCOTT.EMP SCOTT_EMP;
```

NOCACHE

The NOCACHE hint specifies that the blocks retrieved for this table are placed at the least recently used end of the LRU list in the buffer cache when a full table scan is performed. This is the normal behavior of blocks in the buffer cache. The syntax of this hint is:

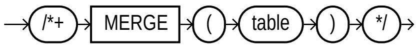


The following example illustrates the NOCACHE hint:

```
SELECT /*+ FULL(SCOTT_EMP) NOCACHE(SCOTT_EMP) */ ENAME
  FROM SCOTT.EMP SCOTT_EMP;
```

MERGE

Merge a view on a per-query basis by using the MERGE hint. The syntax of this hint is:

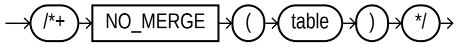


For example:

```
SELECT /*+ MERGE(V) */ T1.X, V.AVG_Y
FROM T1
    (SELECT X, AVG(Y) AS AVG_Y
    FROM T2
    GROUP BY X) V
WHERE T1.X = V.X AND T1.Y = 1;
```

NO_MERGE

The NO_MERGE hint causes Oracle not to merge mergeable views. The syntax of the NO_MERGE hint is:



This hint allows the user to have more influence over the way in which the view will be accessed. For example,

```
SELECT /*+ NO_MERGE(V) */ T1.X, V.AVG_Y
FROM T1
    (SELECT X, AVG(Y) AS AVG_Y
    FROM T2
    GROUP BY X) V
WHERE T1.X = V.X AND T1.Y = 1;
```

causes view *v* not to be merged.

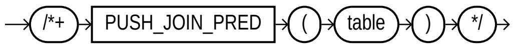
When the NO_MERGE hint is used without an argument, it should be placed in the view query block. When NO_MERGE is used with the view name as an argument, it should be placed in the surrounding query.

This hint is useful if you are using distributed query optimization. For more information about this, please refer to *Oracle8i Distributed Database Systems.*

PUSH_JOIN_PRED

Use the PUSH_JOIN_PRED hint to force pushing of a join predicate into the view.

The syntax of this hint is:

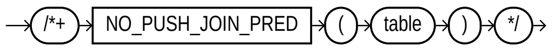


For example:

```
SELECT /*+ PUSH_JOIN_PRED(V) */ T1.X, V.Y
FROM T1
    (SELECT T2.X, T3.Y
    FROM T2, T3
    SHERE T2.X = T3.X) V
where t1.x = v.x and t1.y = 1;
```

NO_PUSH_JOIN_PRED

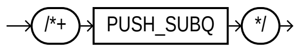
Use the NO_PUSH_JOIN_PRED hint to prevent pushing of a join predicate into the view. The syntax of this hint is:



PUSH_SUBQ

The PUSH_SUBQ hint causes nonmerged subqueries to be evaluated at the earliest possible place in the execution plan. Normally, subqueries that are not merged are executed as the last step in the execution plan. If the subquery is relatively inexpensive and reduces the number of rows significantly, it will improve performance to evaluate the subquery earlier.

The hint has no effect if the subquery is applied to a remote table or one that is joined using a merge join. The syntax of this hint is:

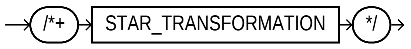


STAR_TRANSFORMATION

The STAR_TRANSFORMATION hint makes the optimizer use the best plan in which the transformation has been used. Without the hint, the optimizer could make a cost-based decision to use the best plan generated without the transformation, instead of the best plan for the transformed query.

Even if the hint is given, there is no guarantee that the transformation will take place. The optimizer will only generate the subqueries if it seems reasonable to do so. If no subqueries are generated, there is no transformed query, and the best plan for the untransformed query will be used regardless of the hint.

The syntax of this hint is:



See Also: *Oracle8i Concepts* for a full discussion of star transformation. Also, the *Oracle8i Reference* describes STAR_TRANSFORMATION_ENABLED; this parameter causes the optimizer to consider performing a star transformation.

ORDERED_PREDICATES

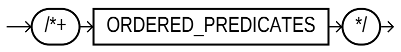
The ORDERED_PREDICATES hint forces the optimizer to preserve the order of predicate evaluation, except for predicates used as index keys. Use this hint in the WHERE clause of SELECT statements.

If you do not use the ORDERED_PREDICATES hint, Oracle evaluates all predicates in the order specified by the following rules. Predicates:

- Without user-defined functions, type methods, or subqueries are evaluated first, in the order specified in the WHERE clause.
- With user-defined functions and type methods that have user-computed costs are evaluated next, in increasing order of their cost.
- With user-defined functions and type methods without user-computed costs are evaluated next, in the order specified in the WHERE clause.
- Not specified in the WHERE clause (for example, predicates transitively generated by the optimizer) are evaluated next.
- With subqueries are evaluated last in the order specified in the WHERE clause.

Note: As mentioned, you cannot use the ORDERED_PREDICATES hint to preserve the order of predicate evaluation on index keys.

The syntax of this hint is:



> **See Also:** *Oracle8i Concepts* for a full discussion of ordered predicates.

Using Hints with Views

Oracle does not encourage you to use hints inside or on views (or subqueries). This is because you can define views in one context and use them in another. However, such hints can result in unexpected plans. In particular, hints inside views or on views are handled differently depending on whether the view is mergeable into the top-level query.

Should you decide, nonetheless, to use hints with views, the following sections describe the behavior in each case.

- Hints and Mergeable Views
- Hints and Nonmergeable Views

Hints and Mergeable Views

This section describes hint behavior with mergeable views.

Optimization Approaches and Goal Hints Optimization approach and goal hints can occur in a top-level query or inside views.

- If there is such a hint in the top-level query, that hint is used regardless of any such hints inside the views.
- If there is no top-level optimizer mode hint, then mode hints in referenced views are used as long as all mode hints in the views are consistent.
- If two or more mode hints in the referenced views conflict, then all mode hints in the views are discarded and the session mode is used, whether default or user-specified.

Access Method and Join Hints on Views Access method and join hints on referenced views are ignored unless the view contains a single table (or references another view with a single table). For such single-table views, an access method hint or a join hint on the view applies to the table inside the view.

Access Method and Join Hints Inside Views Access method and join hints can appear in a view definition.

- If the view is a subquery (that is, if it appears in the FROM clause of a SELECT statement), then all access method and join hints inside the view are preserved when the view is merged with the top-level query.
- For views that are not subqueries, access method and join hints in the view are preserved only if the top-level query references no other tables or views (that is, if the FROM clause of the SELECT statement contains only the view).

Parallel Execution Hints on Views PARALLEL, NOPARALLEL, PARALLEL_INDEX and NOPARALLEL_INDEX hints on views are always recursively applied to all the tables in the referenced view. Parallel execution hints in a top-level query override such hints inside a referenced view.

Hints and Nonmergeable Views

With non-mergeable views, optimization approach and goal hints inside the view are ignored: the top-level query decides the optimization mode.

Since non-mergeable views are optimized separately from the top-level query, access method and join hints inside the view are always preserved. For the same reason, access method hints on the view in the top-level query are ignored.

However, join hints on the view in the top-level query are preserved since, in this case, a non-mergeable view is similar to a table.

8 Tuning Distributed Queries

Oracle supports transparent distributed queries to access data from multiple databases. It also provides many other distributed features, such as transparent distributed transactions and a transparent, fully automatic two-phase commit. This chapter explains how the Oracle8 optimizer decomposes SQL statements, and how this affects performance of distributed queries. The chapter provides guidelines on how to influence the optimizer and avoid performance bottlenecks.

Topics include:

Remote and Distributed Queries

If a SQL statement references one or more remote tables, the optimizer first determines whether all remote tables are located at the same site. If all tables are located at the same remote site, Oracle sends the entire query to the remote site for execution. The remote site sends the resulting rows back to the local site. This is called a remote SQL statement. If the tables are located at more than one site, the optimizer decomposes the query into separate SQL statements to access each of the remote tables. This is called a distributed SQL statement. The site where the query is executed, called the "driving site," is normally the local site.

This section describes:

- Distributed SQL Statements
- EXPLAIN PLAN and SQL Decomposition
- Partition Views

Remote Data Dictionary Information

If a SQL statement references multiple tables, then the optimizer must determine which columns belong to which tables before it can decompose the SQL statement. For example, for this query:

```
SELECT DNAME, ENAME
  FROM DEPT, EMP@REMOTE
 WHERE DEPT.DEPTNO = EMP.DEPTNO
```

The optimizer must first determine that the DNAME column belongs to the DEPT table and the ENAME column to the EMP table. Once the optimizer has the data dictionary information of all remote tables, it can build the decomposed SQL statements.

Column and table names in decomposed SQL statements appear between double quotes. You must enclose in double quotes any column and table names that contain special characters, reserved words, or spaces.

This mechanism also replaces an asterisk (*) in the select list with the actual column names. For example:

```
SELECT * FROM DEPT@REMOTE;
```

Results in the decomposed SQL statement

```
SELECT A1."DEPTNO", A1."DNAME", A1."LOC" FROM "DEPT" A1;
```

Note: For simplicity, double quotes are not used in the remainder of this chapter.

Remote SQL Statements

If the entire SQL statement is sent to the remote database, the optimizer uses table aliases A1, A2, and so on, for all tables and columns in the query, in order to avoid possible naming conflicts. For example:

```
SELECT DNAME, ENAME
  FROM DEPT@REMOTE, EMP@REMOTE
```

```
   WHERE DEPT.DEPTNO = EMP.DEPTNO;
```

is sent to the remote database as:

```
   SELECT A2.DNAME, A1.ENAME
     FROM DEPT A2, EMP A1
   WHERE A1.DEPTNO = A2.DEPTNO;
```

Distributed SQL Statements

When a query accesses data on one or more databases, one site "drives" the execution of the query. This is known as the "driving site"; it is here that the data is joined, grouped and ordered. By default, the local Oracle server is the driving site. A hint called DRIVING_SITE enables you to manually specify the driving site.

The decomposition of SQL statements is important because it determines the number of records or even tables that must be sent through the network. A knowledge of how the optimizer decomposes SQL statements can help you achieve optimum performance for distributed queries.

If a SQL statement references one or more remote tables, the optimizer must decompose the SQL statement into separate queries to be executed on the different databases. For example:

```
   SELECT DNAME, ENAME
   FROM DEPT, EMP@REMOTE
   WHERE DEPT.DEPTNO = EMP.DEPTNO;
```

Might be decomposed into

```
   SELECT DEPTNO, DNAME FROM DEPT;
```

Which is executed locally, and

```
   SELECT DEPTNO, ENAME FROM EMP;
```

which is sent to the remote database. The data from both tables is joined locally. All this is done automatically and transparently for the user or application.

In some cases, however, it might be better to send the local table to the remote database and join the two tables on the remote database. This can be achieved either by creating a view, or by using the DRIVING_SITE hint. If you decide to create a view on the remote database, a database link from the remote database to the local database is also needed.

For example (on the remote database):

```
CREATE VIEW DEPT_EMP AS
SELECT DNAME, ENAME
FROM DEPT@LOCAL, EMP
WHERE DEPT.DEPTNO = EMP.DEPTNO;
```

Then select from the remote view instead of the local and remote tables

```
SELECT * FROM DEPT_EMP@REMOTE;
```

Now the local DEPT table is sent through the network to the remote database, joined on the remote database with the EMP table, and the result is sent back to the local database.

> **See Also:** "DRIVING_SITE" on page 7-54 for details about this hint.

Rule-Based Optimization

Rule-based optimization does not have information about indexes for remote tables. It never, therefore, generates a nested loops join between a local table and a remote table with the local table as the outer table in the join. It uses either a nested loops join with the remote table as the outer table or a sort merge join, depending on the indexes available for the local table.

Cost-Based Optimization

Cost-based optimization can consider more execution plans than rule-based optimization. Cost-based optimization knows whether indexes on remote tables are available, and in which cases it would make sense to use them. Cost-based optimization considers index access of the remote tables as well as full table scans, whereas rule-based optimization considers only full table scans.

The particular execution plan and table access that cost-based optimization chooses depends on the table and index statistics. For example, with:

```
SELECT DNAME, ENAME
  FROM DEPT, EMP@REMOTE
 WHERE DEPT.DEPTNO = EMP.DEPTNO
```

The optimizer might choose the local DEPT table as the driving table and access the remote EMP table using an index, so the decomposed SQL statement becomes:

```
SELECT ENAME FROM EMP WHERE DEPTNO = :1
```

This decomposed SQL statement is used for a nested loops operation.

Using Views

If tables are on more than one remote site, it can be more effective to create a view than to use the DRIVING_SITE hint. If not all tables are on the same remote database, the optimizer accesses each remote table separately. For example:

```
SELECT D.DNAME, E1.ENAME, E2.JOB
  FROM DEPT D, EMP@REMOTE E1, EMP@REMOTE E2
 WHERE D.DEPTNO = E1.DEPTNO
   AND E1.MGR = E2.EMPNO;
```

Results in the decomposed SQL statements

```
SELECT EMPNO, ENAME FROM EMP;
```

And

```
SELECT ENAME, MGR, DEPTNO FROM EMP;
```

If you want to join the two EMP tables remotely, you can create a view to accomplish this. Create a view with the join of the remote tables on the remote database. For example (on the remote database):

```
CREATE VIEW EMPS AS
SELECT E1.DEPTNO, E1.ENAME, E2.JOB
  FROM EMP E1, EMP E2
 WHERE E1.MGR = E2.EMPNO;
```

And now select from the remote view instead of the remote tables:

```
SELECT D.DNAME, E.ENAME, E.JOB
  FROM DEPT D, EMPS@REMOTE E
WHERE D.DEPTNO = E.DEPTNO;
```

This results in the decomposed SQL statement

```
SELECT DEPTNO, ENAME, JOB FROM EMPS;
```

Using Hints

In a distributed query, all hints are supported for local tables. For remote tables, however, you can use only join order and join operation hints. (Hints for access methods, parallel hints, and so on, have no effect.) For remote mapped queries, all hints are supported.

See Also: "Hints for Join Orders" on page 7-51 and "Hints for Join Operations" on page 7-52.

EXPLAIN PLAN and SQL Decomposition

EXPLAIN PLAN gives information not only about the overall execution plan of SQL statements, but also about the way in which the optimizer decomposes SQL statements. EXPLAIN PLAN stores information in the PLAN_TABLE table. If remote tables are used in a SQL statement, the OPERATION column will contain the value REMOTE to indicate that a remote table is referenced, and the OTHER column will contain the decomposed SQL statement that will be sent to the remote database. For example:

```
EXPLAIN PLAN FOR SELECT DNAME FROM DEPT@REMOTE
SELECT OPERATION, OTHER FROM PLAN_TABLE

OPERATION OTHER
--------- -------------------------------------
REMOTE    SELECT A1."DNAME" FROM "DEPT" A1
```

Note the table alias and the double quotes around the column and table names.

See Also: Chapter 13, "Using EXPLAIN PLAN".

Partition Views

You can use partition views to coalesce tables that have the same structure, but that also contain different partitions of data. This is useful for a distributed database where each partition resides on a database and the data in each partition has common geographical properties.

When a query is executed on such a partition view, and the query contains a predicate that contains the result set to a subset of the view's partitions, the optimizer chooses a plan which skips partitions that are not needed for the query. This partition elimination takes place at run time, when the execution plan references all partitions.

Rules for Use

This section describes the circumstances under which a UNION ALL view enables the optimizer to skip partitions. The Oracle server that contains the partition view must conform to the following rules:

- The PARTITION_VIEW_ENABLED initialization parameter is set to TRUE
- The cost-based optimizer is used

> **Note:** To use the cost-based optimizer you must analyze all tables used in the UNION ALL views. Alternatively, you can use a hint or set the parameter OPTIMIZER_MODE to ALL_ROWS or FIRST_ROW. To set OPTIMIZER_MODE or PARTITION_VIEW_ENABLED you can also use the ALTER SESSION statement.

Within a UNION ALL view there are multiple select statements, and each of these is called a "branch". A UNION ALL view is a partition view if each select statement it defines conforms to the following rules:

- The branch has exactly one table in the FROM clause.
- The branch contains a WHERE clause that defines the subset of data from the partition that is contained in the view.
- None of the following are used within the branch: WHERE clause with subquery, group by, aggregate functions, distinct, rownum, connect by/start with.
- The SELECT list of each branch is *, or explicit expansion of "*".
- The column names and column datatypes for all branches in the UNION ALL view are exactly the same.
- All tables used in the branch must have indexes (if any) on the same columns and number of columns.

Partition elimination is based on column transitivity with constant predicates. The WHERE clause used in the query that accesses the partition view is pushed down to the WHERE clause of each of the branches in the UNION ALL view definition. Consider the following example:

```
SELECT * FROM EMP_VIEW WHERE deptno=30;
```

When the view EMP_VIEW is defined as:

```
SELECT * FROM EMP@d10 WHERE deptno=10
  UNION ALL
SELECT * FROM EMP@d20 WHERE deptno=20
  UNION ALL
SELECT * FROM EMP@d30 WHERE deptno=30
  UNION ALL
SELECT * FROM EMP@d40 WHERE deptno=40
```

The "WHERE deptno=30" predicate used in the query is pushed down to the queries in the UNION ALL view. For a WHERE clause such as "WHERE deptno=10 and deptno=30", the optimizer applies transitivity rules to generate an extra predicate of "10=30". This extra predicate is always false, thus the table (EMP@d10) need not be accessed.

Transitivity applies to predicates which conform to the following rules:

- The predicates in the WHERE clause for each branch are of the form:

  ```
  RELATION AND RELATION ...
  ```

  where relation is of the form

  ```
  COLUMN_NAME RELOP CONSTANT_EXPRESSION
  ```

  and relop is one of =, !=, >, >=, <, <=

  Note: BETWEEN ... AND is allowed by these rules, but IN is not.

- At least one predicate in the query referencing the view exists in the same form.

EXPLAIN PLAN Output

To confirm that the system recognizes a partition view, check the EXPLAIN PLAN output. The following operations will appear in the OPERATIONS column of the EXPLAIN PLAN output, if a query was executed on a partition view:

VIEW	This entry should include the optimizer cost in the COST column.
UNION-ALL	This entry should specify PARTITION in the OPTION column.

FILTER	When an operation is a child of the UNION-ALL operation, FILTER indicates that a constant predicate was generated that will always be FALSE. The partition will be eliminated.

If PARTITION does not appear in the option column of the UNION-ALL operation, the partition view was not recognized, and no partitions were eliminated. Make sure that the UNION ALL view adheres to the rules as defined in "Rules for Use" on page 8-7.

Partition View Example

The following example shows a partition view CUSTOMER that is partitioned into two partitions. The EAST database contains the East Coast customers, and the WEST database contains the customers from the West Coast.

The WEST database contains the following table CUSTOMER_WEST:

```
CREATE TABLE CUSTOMER_WEST
( cust_no   NUMBER CONSTRAINT CUSTOMER_WEST_PK PRIMARY KEY,
  cname     VARCHAR2(10),
  location  VARCHAR2(10)
 );
```

The EAST database contains the database CUSTOMER_EAST:

```
CREATE TABLE CUSTOMER_EAST
( cust_no   NUMBER CONSTRAINT CUSTOMER_EAST_PK PRIMARY KEY,
  cname     VARCHAR2(10),
  location  VARCHAR2(10)
 );
```

The following partition view is created at the EAST database (you could create a similar view at the WEST database):

```
CREATE VIEW customer AS
    SELECT * FROM CUSTOMER_EAST
    WHERE location='EAST'
    UNION ALL
    SELECT * FROM CUSTOMER_WEST@WEST
    WHERE location='WEST';
```

If you execute the following statement, notice that the CUSTOMER_WEST table in the WEST database is not accessed:

```
EXPLAIN PLAN FOR SELECT * FROM CUSTOMER WHERE location='EAST';
```

Note: The EAST database still needs column name and column datatype information for the CUSTOMER_WEST table, therefore it still needs a connection to the WEST database. In addition that the cost-based optimizer must be used. You could do this, for example, by issuing the statement ALTER SESSION SET OPTIMIZER_MODE=ALL_ROWS.

As shown in the EXPLAIN PLAN output, the optimizer recognizes that the CUSTOMER_WEST partition need not be accessed:

```
    SELECT LPAD(' ',LEVEL*3-3)||OPERATION OPERATION,COST,OPTIONS,
    OBJECT_NODE, OTHER
    FROM PLAN_TABLE
    CONNECT BY PARENT_ID = PRIOR ID
    START WITH PARENT_ID IS NULL

OPERATION                 COST OPTIONS    OBJECT_NOD OTHER
------------------------- ---- ---------- ---------- -------------------------
SELECT STATEMENT             1
   VIEW                      1
      UNION-ALL                PARTITION
         TABLE ACCESS        1 FULL
         FILTER
            REMOTE           1            WEST.WORLD SELECT "CUST_NO","CNAME",
                                                     "LOCATION" FROM "CUSTOMER
                                                     _WEST" "CUSTOMER_WEST" WH
                                                     ERE "LOCATION"='EAST' AND
                                                      "LOCATION"='WEST'
```

Distributed Query Restrictions

Distributed queries within the same version of Oracle have these restrictions:

- Cost-based optimization should be used for distributed queries. Rule-based optimization does not generate nested loop joins between remote and local tables when the tables are joined with equijoins.
- In cost-based optimization, no more than 20 indexes per remote table are considered when generating query plans. The order of the indexes varies; if the 20-index limitation is exceeded, random variation in query plans may result.

- Reverse indexes on remote tables are not visible to the optimizer. This can prevent nested-loop joins from being used for remote tables if there is an equijoin using a column with only a reverse index.
- Cost-based optimization cannot recognize that a remote object is partitioned. Thus the optimizer may generate less than optimal plans for remote partitioned objects, particularly when partition pruning would have been possible, had the object been local.
- Remote views are not merged and the optimizer has no statistics for them. It is best to replicate all mergeable views at all sites to obtain good query plans. (See the following exception.)
- Neither cost-based nor rule-based optimization can execute joins remotely. All joins are executed at the driving site. This can affect performance for CREATE TABLE ... AS SELECT if all the tables in the select list are remote. In this case you should create a view for the SELECT statement at the remote site.

Transparent Gateways

The Transparent Gateways are used to access data from other data sources (relational databases, hierarchical databases, file systems, and so on). Transparent Gateways provide a means to transparently access data from a non-Oracle system, just as if it were another Oracle database.

Optimizing Heterogeneous Distributed SQL Statements

When a SQL statement accesses data from non-Oracle systems, it is said to be a heterogeneous distributed SQL statement. To optimize heterogeneous distributed SQL statements, follow the same guidelines as for optimizing distributed SQL statements that access Oracle databases only. However, you must take into consideration that the non-Oracle system usually does not support all the functions and operators that Oracle8 supports. The Transparent Gateways therefore tell Oracle (at connect time) which functions and operators they do support. If the other data source does not support a function or operator, Oracle will perform that function or operator. In this case Oracle obtains the data from the other data source and applies the function or operator locally. This affects the way in which the SQL statements are decomposed and can affect performance, especially if Oracle is not on the same machine as the other data source.

Gateways and Partition Views

You can use partition views with Oracle Transparent Gateways version 8 or higher. Make sure you adhere to the rules that are defined in "Rules for Use" on page 8-7. In particular:

- The cost-based optimizer must be used, by using hints or setting the parameter OPTIMIZER_MODE to ALL_ROWS or FIRST_ROWS.
- Indexes used for each partition must be the same. Please consult your gateway specific installation and users guide to find out whether the gateway will send index information of the non-Oracle system to the Oracle Server. If the gateway will send index information to the optimizer, make sure that each partition uses the same number of indexes, and that you have indexed the same columns. If the gateway does not send index information, the Oracle optimizer will not be aware of the indexes on partitions. Indexes are therefore considered to be the same for each partition in the non-Oracle system. If one partition resides on an Oracle server, you cannot have an index defined on that partition.
- The column names and column datatypes for all branches in the UNION ALL view must be the same. Non-Oracle system datatypes are mapped onto Oracle datatypes. Make sure that the datatypes of each partition that resides in the different non-Oracle systems all map to the same Oracle datatype. To see how datatypes are mapped onto Oracle datatypes, you can execute a DESCRIBE command in SQL*Plus or Server Manager.

Summary: Optimizing Performance of Distributed Queries

You can improve performance of distributed queries in several ways:

- Choose the best SQL statement

  In many cases there are several SQL statements which can achieve the same result. If all tables are on the same database, the difference in performance between these SQL statements might be minimal; but if the tables are located on different databases, the difference in performance might be more significant.

- Use cost-based optimization

  Cost-based optimization can use indexes on remote tables, considers more execution plans than rule-based optimization, and generally gives better results. With cost-based optimization performance of distributed queries is generally satisfactory. Only in rare occasions is it necessary to change SQL statements, create views, or use procedural code.

- Use views

In some situations, views can be used to improve performance of distributed queries; for example:

- To join several remote tables on the remote database
- To send a different table through the network
- Use procedural code

In some rare occasions it can be more efficient to replace a distributed query by procedural code, such as a PL/SQL procedure or a precompiler program. This option is mentioned here only for completeness, not because it is often needed.

9 Transaction Modes

This chapter describes the different modes in which read consistency is performed. Topics in this chapter include:

Using Discrete Transactions

You can improve the performance of short, nondistributed transactions by using the BEGIN_DISCRETE_TRANSACTION procedure. This procedure streamlines transaction processing so short transactions can execute more rapidly. This section describes:

Deciding When to Use Discrete Transactions

Discrete transaction processing is useful for transactions that:

- Modify only a few database blocks
- Never change an individual database block more than once per transaction
- Do not modify data likely to be requested by long-running queries

- Do not need to see the new value of data after modifying the data
- Do not modify tables containing any LONG values

In deciding to use discrete transactions, you should consider the following factors:

- Can the transaction be designed to work within the constraints placed on discrete transactions, as described in "Usage Notes" on page 9-3?
- Does using discrete transactions result in a significant performance improvement under normal usage conditions?

Discrete transactions can be used concurrently with standard transactions. Choosing whether to use discrete transactions should be a part of your normal tuning procedure. Discrete transactions can be used only for a subset of all transactions, for sophisticated users with advanced application requirements. However, where speed is the most critical factor, the performance improvements can justify the design constraints.

How Discrete Transactions Work

During a discrete transaction, all changes made to any data are deferred until the transaction commits. Redo information is generated, but is stored in a separate location in memory.

When the transaction issues a commit request, the redo information is written to the redo log file (along with other group commits) and the changes to the database block are applied directly to the block. The block is written to the database file in the usual manner. Control is returned to the application once the commit completes. Oracle does not need to generate undo information because the block is not actually modified until the transaction is committed and the redo information is stored in the redo log buffers.

As with other transactions, the uncommitted changes of a discrete transaction are not visible to concurrent transactions. For regular transactions, undo information is used to re-create old versions of data for queries that require a consistent view of the data. Because no undo information is generated for discrete transactions, a discrete transaction that starts and completes during a long query can cause the query to receive the "snapshot too old" error if the query requests data changed by the discrete transaction. For this reason, you might want to avoid performing queries that access a large subset of a table that is modified by frequent discrete transactions.

Errors During Discrete Transactions

Any errors encountered during processing of a discrete transaction cause the predefined exception DISCRETE_TRANSACTION_FAILED to be raised. These errors include the failure of a discrete transaction to comply with the usage notes outlined below. (For example, calling BEGIN_DISCRETE_TRANSACTION after a transaction has begun, or attempting to modify the same database block more than once during a transaction, raises the exception.)

Usage Notes

The BEGIN_DISCRETE_TRANSACTION procedure must be called before the first statement in a transaction. This call to the procedure is effective only for the duration of the transaction (that is, once the transaction is committed or rolled back, the next transaction is processed as a standard transaction).

Transactions that use this procedure cannot participate in distributed transactions.

Although discrete transactions cannot see their own changes, you can obtain the old value and lock the row, using the FOR UPDATE clause of the SELECT statement, before updating the value.

Because discrete transactions cannot see their own changes, a discrete transaction cannot perform inserts or updates on both tables involved in a referential integrity constraint.

For example, assume the EMP table has a FOREIGN KEY constraint on the DEPTNO column that refers to the DEPT table. A discrete transaction cannot attempt to add a department into the DEPT table and then add an employee belonging to that department, because the department is not added to the table until the transaction commits and the integrity constraint requires that the department exist before an insert into the EMP table can occur. These two operations must be performed in separate discrete transactions.

Because discrete transactions can change each database block only once, some combinations of data manipulation statements on the same table are better suited for discrete transactions than others. One INSERT statement and one UPDATE statement used together are the least likely to affect the same block. Multiple UPDATE statements are also unlikely to affect the same block, depending on the size of the affected tables. Multiple INSERT statements (or INSERT statements that use queries to specify values), however, are likely to affect the same database block. Multiple DML operations performed on separate tables do not affect the same database blocks, unless the tables are clustered.

Example

An application for checking out library books is an example of a transaction type that uses the BEGIN_DISCRETE_TRANSACTION procedure. The following procedure is called by the library application with the book number as the argument. This procedure checks to see if the book is reserved before allowing it to be checked out. If more copies of the book have been reserved than are available, the status RES is returned to the library application, which calls another procedure to reserve the book, if desired. Otherwise, the book is checked out and the inventory of books available is updated.

```
CREATE PROCEDURE checkout (bookno IN NUMBER (10)
                           status OUT VARCHAR(5))
AS
DECLARE
   tot_books   NUMBER(3);
   checked_out NUMBER(3);
   res         NUMBER(3);
BEGIN
   dbms_transaction.begin_discrete_transaction;
   FOR i IN 1 .. 2 LOOP
      BEGIN
         SELECT total, num_out, num_res
           INTO tot_books, checked_out, res
           FROM books
           WHERE book_num = bookno
           FOR UPDATE;
      IF res >= (tot_books - checked_out)
      THEN
         status := ’RES’;
      ELSE
         UPDATE books SET num_out = checked_out + 1
            WHERE book_num = bookno;
         status := ’AVAIL’
      ENDIF;
      COMMIT;
      EXIT;
   EXCEPTION
      WHEN dbms_transaction.discrete_transaction_failed THEN
         ROLLBACK;
      END;
   END LOOP;
END;
```

For the above loop construct, if the DISCRETE_TRANSACTION_FAILED exception occurs during the transaction, the transaction is rolled back, and the loop executes the transaction again. The second iteration of the loop is not a discrete transaction, because the ROLLBACK statement ended the transaction; the next transaction processes as a standard transaction. This loop construct ensures that the same transaction is attempted again in the event of a discrete transaction failure.

Using Serializable Transactions

Oracle allows application developers to set the isolation level of transactions. The isolation level determines what changes the transaction and other transactions can see. The ISO / ANSI SQL3 specification details the following levels of transaction isolation.

SERIALIZABLE	Transactions lose no updates, provide repeatable reads, and do not experience phantoms. Changes made to a serializable transaction are visible only to the transaction itself.
READ COMMITTED	Transactions do not have repeatable reads, and changes made in this transaction or other transactions are visible to all transactions. This is the default transaction isolation.

If you wish to set the transaction isolation level, you must do so before the transaction begins. Use the SET TRANSACTION ISOLATION LEVEL statement for a particular transaction, or the ALTER SESSION SET ISOLATION_LEVEL statement for all subsequent transactions in the session.

See Also: *Oracle8i SQL Reference* for more information on the syntax of SET TRANSACTION and ALTER SESSION.

10 Managing SQL and Shared PL/SQL Areas

Oracle compares SQL statements and PL/SQL blocks issued directly by users and applications as well as recursive SQL statements issued internally by a DDL statement. If two identical statements are issued, the SQL or PL/SQL area used to process the first instance of the statement is *shared*, or used for the processing of the subsequent executions of that same statement.

Shared SQL and PL/SQL areas are shared memory areas; any Oracle process can use a shared SQL area. The use of shared SQL areas reduces memory usage on the database server, thereby increasing system throughput.

Shared SQL and PL/SQL areas age out of the shared pool according to a "least recently used" (LRU) algorithm (similar to database buffers). To improve performance and prevent reparsing, you may want to prevent large SQL or PL/SQL areas from aging out of the shared pool.

This chapter explains the use of shared SQL to improve performance. Topics in this chapter include:

- Comparing SQL Statements and PL/SQL Blocks
- Keeping Shared SQL and PL/SQL in the Shared Pool

Comparing SQL Statements and PL/SQL Blocks

This section describes

- Testing for Identical SQL Statements
- Aspects of Standardized SQL Formatting

Testing for Identical SQL Statements

Oracle automatically notices when two or more applications send identical SQL statements or PL/SQL blocks to the database. It does not have to parse a statement to determine whether it is identical to another statement currently in the shared pool. Oracle distinguishes identical statements using the following steps:

1. The text string of an issued statement is hashed. If the hash value is the same as a hash value for an existing SQL statement in the shared pool, Oracle proceeds to Step 2.
2. The text string of the issued statement, including case, blanks, and comments, is compared to all existing SQL statements that were identified in Step 1.
3. The objects referenced in the issued statement are compared to the referenced objects of all existing statements identified in Step 2. For example, if two users have EMP tables, the statement

   ```
   SELECT * FROM emp;
   ```

   is not considered identical because the statement references different tables for each user.
4. The bind types of bind variables used in a SQL statement must match.

Note: Most Oracle products convert the SQL before passing statements to the database. Characters are uniformly changed to upper case, white space is compressed, and bind variables are renamed so that a consistent set of SQL statements is produced.

Aspects of Standardized SQL Formatting

It is neither necessary nor useful to have every user of an application attempt to write SQL statements in a standardized way. It is unlikely that 300 people writing ad hoc dynamic statements in standardized SQL will generate the same SQL statements. The chances that they will all want to look at exactly the same columns in exactly the same tables in exactly the same order is remote. By contrast, 300 people running the same application—executing command files—*will* generate the same SQL statements.

Within an application there is a very minimal advantage to having 2 statements almost the same, and 300 users using them; there is a major advantage to having one statement used by 600 users.

Keeping Shared SQL and PL/SQL in the Shared Pool

This section describes two techniques of keeping shared SQL and PL/SQL in the shared pool:

- Reserving Space for Large Allocations
- Preventing Objects from Aging Out

Reserving Space for Large Allocations

A problem can occur if users fill the shared pool, and then a large package ages out. If someone should then call the large package back in, a significant amount of maintenance is required to create space for it in the shared pool. You can avoid this problem by reserving space for large allocations with the SHARED_POOL_RESERVED_SIZE initialization parameter. This parameter sets aside room in the shared pool for allocations larger than the value specified by the SHARED_POOL_RESERVED_SIZE_MIN_ALLOC parameter.

Note: Although Oracle uses segmented codes to reduce the need for large areas of contiguous memory, performance may improve if you pin large objects in memory.

Preventing Objects from Aging Out

The DBMS_SHARED_POOL package lets you keep objects in shared memory, so they do not age out with the normal LRU mechanism. The DBMSPOOL.SQL and

PRVTPOOL.PLB procedure scripts create the package specification and package body for DBMS_SHARED_POOL.

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; they do not age out with the normal LRU mechanism. This procedure ensures that memory is available and prevents sudden, inexplicable slowdowns in user response time that occur when SQL and PL/SQL areas are accessed after aging out.

When to Use DBMS_SHARED_POOL

The procedures provided with the DBMS_SHARED_POOL package may be useful when loading large PL/SQL objects, such as the STANDARD and DIUTIL packages.

When large PL/SQL objects are loaded, user response time is affected because of the large number of smaller objects that need to age out of the shared pool to make room. This is due to memory fragmentation. In some cases, there may be insufficient memory to load the large objects.

DBMS_SHARED_POOL is also useful for frequently executed triggers. You may want to keep compiled triggers on frequently used tables in the shared pool.

DBMS_SHARED_POOL also supports sequences. Sequence numbers are lost when a sequence ages out of the shared pool. DBMS_SHARED_POOL is useful for keeping sequences in the shared pool and thus preventing the loss of sequence numbers.

How to Use DBMS_SHARED_POOL

To use the DBMS_SHARED_POOL package to pin a SQL or PL/SQL area, complete the following steps.

1. Decide which packages or cursors to pin in memory.
2. Start up the database.
3. Make the call to DBMS_SHARED_POOL.KEEP to pin it.

This procedure ensures that your system does not run out of the shared memory before the object is loaded. Finally, by pinning the object 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.

The procedures provided with the DBMS_SHARED_POOL package are described below.

DBMS_SHARED_POOL.SIZES

This procedure shows the objects in the shared pool that are larger than the specified size.

```
DBMS_SHARED_POOL.SIZES(MINSIZE IN NUMBER)
```

Input Parameter:

minsize	Display objects in shared pool larger than this size, where size is measured in kilobytes.

To display the results of this procedure, before calling this procedure issue the following command using SQL*Plus:

```
SET SERVEROUTPUT ON SIZE minsize
```

You can use the results of this command as arguments to the KEEP or UNKEEP procedures.

For example, to show the objects in the shared pool that are larger than 2000 kilobytes, issue the following SQL*Plus commands:

```
SET SERVEROUTPUT ON SIZE 2000
EXECUTE DBMS_SHARED_POOL.SIZES(2000);
```

DBMS_SHARED_POOL.KEEP

This procedure lets you keep an object in the shared pool.

```
DBMS_SHARED_POOL.KEEP(OBJECT IN VARCHAR2,
[TYPE IN CHAR DEFAULT P])
```

Input Parameters:

object	Either the parameter name or the cursor address of the object to be kept in the shared pool. This is the value displayed when you call the SIZES procedure.
type	The type of the object to be kept in the shared pool. Types include:

P	package
C	cursor
R	trigger
Q	sequence

When you pin ADT types, the type body as well as the specification is pinned in shared memory so the LRU mechanism does not age them out. You must have execute privilege on a type to be able to pin it in shared memory.

For example, an attempt to keep a type ’TY’ whose body does not exist results in only keeping the type spec TY. Once you create the type body for TY, you must keep the type again to also pin the type body for ’TY’ in shared memory.

The following example demonstrates how user ’user2’ keeps type ’TY’ in user1’s schema pinned in shared memory. This example assumes user2 has execute privilege on user1.TY:

```
BEGIN
 SYS.DBMS_SHARED_POOL.KEEP('user1.TY', 'T');
END;
```

Note: The flag can be either ’T’ or ’t’.

DBMS_SHARED_POOL.UNKEEP

This procedure allows an object that you have pinned in the shared pool to age out of the shared pool.

```
DBMS_SHARED_POOL.UNKEEP(OBJECT IN VARCHAR2,
[TYPE IN CHAR DEFAULT P])
```

Input Parameters:

object	Either the parameter name or the cursor address of the object that you no longer want kept in the shared pool. This is the value displayed when you call the SIZES procedure.
type	Type of the object to be aged out of the shared pool. Types include:

P	package
C	cursor
R	trigger
Q	sequence

11 Optimizing Data Warehouse Applications

This chapter introduces integrated Oracle features for tuning enterprise-scale data warehouses. By intelligently tuning the system, the data layout, and the application, you can build a high performance, scalable data warehouse.

Topics in this chapter include:

Characteristics of Data Warehouse Applications

Data warehousing applications process a substantial amount of data by means of many CPU- and I/O-bound, data-intensive tasks such as:

- Loading, indexing, and summarizing tables
- Scanning, joining, sorting, aggregating, and fetching data

The resources required to complete the tasks on many gigabytes of data distinguishes data warehousing applications from other types of data processing. The bulk and complexity of your data may clearly indicate that you need to deploy multiple CPUs, investigate parallel processing, or consider specific data processing features that are directly relevant to the tasks at hand.

For example, in a typical data warehousing application, data-intensive tasks might involve 100 or more gigabytes of data. At a processing speed of 2GB to 30GB of data per hour per CPU, a single CPU might need several hours to perform this size of a task.

With more than a single gigabyte of data, and certainly with upwards of 10G, you need to consider increasing the number of CPUs.

Similarly, if you need to copy 10 gigabytes of data, consider that using Export/Import might take a single CPU several hours. By contrast, using parallel CREATE TABLE . . . AS SELECT on 10 CPUs or more might take less than an hour.

Actual processing time depends on many factors, such as the complexity of the queries, the processing speed to which a particular hardware configuration can be tuned, and so on. Always run simple tests on your own system to find out its performance characteristics with regard to particular operations.

Building a Data Warehouse

This section briefly describes features useful for building a data warehouse. It includes:

- Materialized Views and Dimensions
- Parallel CREATE TABLE . . . AS SELECT
- Parallel Index Creation
- Fast Full Index Scan
- Partitioned Tables
- ANALYZE Statement
- Parallel Load
- Constraints

See Also: *Oracle8i Concepts* and *Oracle8i SQL Reference*.

Materialized Views and Dimensions

Materialized views are stored summaries of queries containing precomputed results. Materialized views greatly improve data warehouse query processing. Dimensions describe the relationships among data and are required for performing more complex types of query rewrites.

See Also: For more information about materialized views, please see *Section VI, "Materialized Views"*, *Oracle8i Concepts*, the *Oracle8i Administrator's Guide*, and the *Oracle8i Supplied Packages Reference*.

Parallel CREATE TABLE . . . AS SELECT

The ability to perform CREATE TABLE . . . AS SELECT operations in parallel enables you to reorganize extremely large tables efficiently. You might find it prohibitive to take a serial approach to reorganizing or reclaiming space in a table containing tens or thousands of gigabytes of data. Using Export/Import to perform such a task might result in an unacceptable amount of downtime. If you have a lot of temporary space available, you can use CREATE TABLE . . . AS SELECT to perform such tasks in parallel. With this approach, redefining integrity constraints is optional. This feature is often used for creating summary tables, which are precomputed results stored in the data warehouse.

See Also: "Creating and Populating Tables in Parallel" on page 27-18 and *Oracle8i Concepts*.

Parallel Index Creation

The ability to create indexes in parallel benefits both data warehousing and OLTP applications. On extremely large tables, rebuilding an index may take a long time. Periodically DBAs may load a large amount of data and rebuild the index. With the ability to create indexes in parallel, you may be able to drop an index before loading new data, and re-create it afterwards.

See Also: "Creating Indexes in Parallel" on page 26-71.

Fast Full Index Scan

FAST FULL SCAN on the index is a faster alternative to a full table scan when an existing index already contains all the columns that are needed for the query. It can use multiblock I/O and can be parallelized just like a table scan. The hint INDEX_FFS enforces fast full index scan.

See Also: "Using Fast Full Index Scans" on page 6-8 and "INDEX_FFS" on page 7-48.

Partitioned Tables

You can avoid downtime with very large or mission-critical tables by using partitions. You can divide a large table into multiple physical tables using partitioning criteria. In a data warehouse you can manage historical data by partitioning by date. You can then perform on a partition level all of the operations you might normally perform on the table level, such as backup and restore. You can add space for new data by adding a new partition, and delete old data by dropping an existing partition. Queries that use a key range to select from a partitioned table retrieve only the partitions that fall within that range. In this way partitions offer significant improvements in availability, administration and table scan performance.

Note: For performance reasons, in Oracle partitioned tables should be used rather than partition views. Please see *Oracle8i Migration* for instructions on migrating from partition views to partitioned tables.

See Also: "Partitioning Data" on page 26-45 and *Oracle8i Concepts* for information about partitioned tables.

ANALYZE Statement

You can use the ANALYZE statement to analyze the storage characteristics of tables, indexes, and clusters to gather statistics which are then stored in the data dictionary. The optimizer uses these statistics in a cost-based approach to determine the most efficient execution plan for the SQL statements you issue. Statistics can be either computed or estimated, depending on the amount of overhead you are willing to allow for this purpose.

See Also: The *Oracle8i Administrator's Guide.*

Parallel Load

When very large amounts of data must be loaded, the destination table may be unavailable for an unacceptable amount of time. The ability to load data in parallel can dramatically slash the amount of downtime necessary.

See Also: Chapter 26, "Tuning Parallel Execution", especially "Phase Three - Creating, Populating, and Refreshing the Database" on page 26-63 and *Oracle8i Utilities* for a description of SQL*Loader conventional and direct path loads.

Constraints

Validated constraints degrade performance for DML statements, loads, and index maintenance. However, some query optimizations depend on the presence of validated constraints.

The fastest way to move a set of constraints from the DISABLE NOVALIDATED state to ENABLE VALIDATED is to first ENABLE NOVALIDATE them all. Then validate them individually. Validation still requires a long time, but you can query and modify all tables as soon as the ENABLE NOVALIDATE is finished. A direct load automatically re-enables constraints this way.

Data warehouses sometimes benefit from the DISABLE VALIDATE state. This state allows the optimizer to recognize the validity of a unique or primary key, yet it does not require an index. Inserts, updates, and deletes are disallowed on keys in the DISABLE VALIDATE state.

See Also: For more information, please refer to the chapter on "Managing Integrity" in the *Oracle8i Administrator's Guide*. The *Oracle8i SQL Reference*. also contains information about the DISABLE VALIDATE / NOVALIDATE keywords.

Querying a Data Warehouse

This section summarizes Oracle features useful for querying a data warehouse. It includes:

- Oracle Parallel Server Option
- Parallel-Aware Optimizer
- Parallel Execution
- Bitmap Indexes
- Star Queries
- Star Transformation
- Query Rewrites

Oracle Parallel Server Option

The Oracle Parallel Server option provides the following benefits important to both OLTP and data warehousing applications:

- Application failover
- Scalable performance
- Load balancing
- Multi-user scalability

Oracle Parallel Server failover capability (the ability of the application to reconnect automatically if the connection to the database is broken) results in improved availability, a primary benefit for OLTP applications. Likewise, scalability in the number of users that can connect to the database is a major benefit in OLTP environments. OLTP performance on Oracle Parallel Server can scale as well, if an application's data is isolated onto a single server.

For data warehousing applications, scalability of performance is a primary benefit of Oracle Parallel Server. The architecture of Oracle Parallel Server allows parallel execution to perform excellent load balancing of work at runtime. If a node in an Oracle Parallel Server cluster or MPP is temporarily slowed down, work that was originally assigned to parallel execution servers on that node (but not yet commenced by those servers) may be performed by servers on other nodes, hence preventing that node from becoming a serious bottleneck. Even though Oracle Parallel Server is a cornerstone of parallel execution on clusters and MPPs, in a mostly query environment the overhead on the distributed lock manager is minimal.

See Also: Please refer to the *Oracle8i Parallel Server Concepts and Administration* text.

Parallel-Aware Optimizer

Knowledge about parallelism is incorporated into the Oracle cost-based optimizer. Parallel execution considerations are thus a fundamental component in arriving at query execution plans. In addition, you can control the trade-off of throughput for response time in plan selection.

The optimizer chooses intelligent defaults for the degree of parallelism based on available processors and the number of disk drives storing data the query will access. Access path choices (such as table scans vs. index access) take into account the degree of parallelism, resulting in plans that are optimized for parallel execution. Execution plans are more scalable, and there is improved correlation between optimizer cost and execution time for parallel execution.

The initialization parameter OPTIMIZER_PERCENT_PARALLEL defines the weighting that the optimizer uses to minimize response time in its cost functions.

See Also: "OPTIMIZER_PERCENT_PARALLEL" on page 26-23.

Parallel Execution

Parallel execution enables multiple processes to simultaneously process a single query or DML statement. By dividing the task among multiple server processes, Oracle executes the operation more quickly than if only one server process were used.

Parallel execution dramatically improves performance for data-intensive data warehousing operations. It helps systems scale in performance when adding hardware resources. The greatest performance benefits are on SMP (Symmetric Multiprocessing), clustered, or MPP (Massively Parallel Platforms) where query processing can be effectively spread out among many CPUs on a single system.

See Also: Chapter 26, "Tuning Parallel Execution", Chapter 27, "Understanding Parallel Execution Performance Issues", and *Oracle8i Concepts* for more details on parallel execution.

Bitmap Indexes

Regular B*-tree indexes work best when each key or key range references only a few records, such as employee names. Bitmap indexes, by contrast, work best when each key references many records, such as employee gender.

Bitmap indexing provides the same functionality as regular indexes but uses a different internal representation that makes it very fast and space efficient. Bitmap indexing benefits data warehousing applications with large amounts of data queried on an ad hoc basis but with a low level of concurrent transactions.

Bitmap indexes reduce the response time for many types of ad hoc queries. They also offer considerably reduced space usage compared to other indexing techniques and dramatic performance gains even on very low-end hardware. You can create bitmap indexes in parallel; they are completely integrated with cost-based optimization.

Domain Indexes

You can create an application-specific index, known as a "domain index," by using the indextype schema object. Domain indexes are used for indexing data in application-specific domains. A domain index is an instance of an index that is created, managed, and accessed by routines supplied by an indextype. This is in contrast to B*-tree indexes which are maintained by the database and are referred to as "indexes".

See Also: "Using Domain Indexes" on page 6-23.

Star Queries

One type of data warehouse design is known as a "star" schema. This typically consists of one or more very large "fact" tables and a number of much smaller "dimension" or reference tables. A star query is one that joins several of the dimension tables, usually by predicates in the query, to one of the fact tables.

Oracle cost-based optimization recognizes star queries and generates efficient execution plans for them; indeed, you *must* use cost-based optimization to get efficient star query execution. To enable cost-based optimization, ANALYZE your tables and make sure not to set the OPTIMIZER_MODE parameter to RULE.

See Also: *Oracle8i Concepts* regarding optimization of star queries and "STAR" for information about the STAR hint.

Star Transformation

Star transformation is a cost-based transformation designed to execute star queries efficiently. Whereas star optimization works well for schemas with a small number of dimensions and dense fact tables, star transformation works well for schemas with a large number of dimensions and sparse fact tables.

Star transformation is enabled by setting the initialization parameter STAR_TRANSFORMATION_ENABLED to TRUE. You can use the STAR_TRANSFORMATION hint to make the optimizer use the best plan in which the transformation has been used.

See Also: For more information, the discussion under the heading "STAR_TRANSFORMATION" on page 7-66 explains how to use this hint. Also refer to *Oracle8i Concepts* for a full discussion of star transformation. *Oracle8i Reference* describes the STAR_TRANSFORMATION_ENABLED initialization parameter.

Query Rewrites

If you have defined materialized views, Oracle can transparently rewrite queries to use summary tables rather than detail tables.

Tuning Data Warehouse Applications

Tuning data warehouse applications involves both serial and parallel SQL statement tuning.

Shared SQL is not recommended with data warehousing applications. Use literal values in these SQL statements, rather than bind variables. If you use bind variables, the optimizer will make a blanket assumption about the selectivity of the column. If you specify a literal value, by contrast, the optimizer can use value histograms and so provide a better access plan.

> **See Also:** Chapter 10, "Managing SQL and Shared PL/SQL Areas".

Backup and Recovery of the Data Warehouse

Recoverability has various levels: recovery from disk failure, human error, software failure, fire, and so on, requires different procedures. Oracle provides only part of the solution. Organizations must decide how much to spend on backup and recovery by considering the business cost of a long outage.

The NOLOGGING option enables you to perform certain operations without the overhead of generating a log. Even without logging, you can avoid disk failure if you use disk mirroring or RAID technology. If you load your warehouse from tapes every day or week, you might satisfactorily recover from all failures simply by saving copies of the tape in several remote locations and reloading from tape when something goes wrong.

At the other end of the spectrum, you could both mirror disks and take backups and archive logs, and maintain a remote standby system. The mirrored disks prevent loss of availability for disk failure, and also protect against total loss in the event of human error (such as dropping the wrong table) or software error (such as disk block corruption). In the event of fire, power failure, or other problems at the primary site, the backup site prevents long outages.

> **See Also:** For more information on recovery and the NOLOGGING option, see "[NO]LOGGING Option" on page 26-76, the *Oracle8i Administrator's Guide* and *Oracle8i SQL Reference.*

Tuning Fast-start Parallel Recovery

Fast-start Parallel Recovery improves recovery throughput and decreases the time required to apply rollback segments by using multiple, concurrent processes. The initialization parameter FAST_START_PARALLEL_ROLLBACK determines the upper limit for the degree of parallel recovery used.

Fast-start Parallel Recovery automatically starts enough recovery processes to have at least one process for each unrecovered rollback segment. When possible, this feature assigns multiple processes to roll back a single transaction. However, the number of active recovery processes never exceeds the upper limit set by the value for FAST_START_PARALLEL_ROLLBACK.

When to Use Fast-start Parallel Recovery

Typically, DSS environments are more likely to have large transactions than OLTP environments. Therefore, Fast-start Parallel Recovery is more often applicable to DSS systems.

For any instance, determine how aggressively to deploy Fast-start Parallel Recovery by monitoring your transaction recovery progress. Do this by examining the contents of two recovery-related V$ views as explained later in this section.

Determining Adequate Parallelism for Fast-start Parallel Recovery

The default setting of 20 for FAST_START_PARALLEL_ROLLBACK is adequate for most systems. However, you can more accurately tune this parameter using information in the V$FAST_START_SERVERS and V$FAST_START_TRANSACTIONS views as described under the following headings. Also consider your overall recovery processing goals.

V$FAST_START_SERVERS View

This view provides data about the progress of server processes performing Fast-start Parallel Recovery. Enable Fast-start Parallel Recovery by setting FAST_START_PARALLEL_ROLLBACK to a value of 1 or greater. When enabled, SMON recovers transactions serially and V$FAST_START_SERVERS has only one row.

When you enable Fast-start Parallel Recovery, the total number of rows in this view is one more than the number of active recovery processes. The additional process is SMON. The value for PARALLEL_TRANSACTION_RECOVERY_DEGREE limits the number of rows in this view. Each row in the view corresponds to a recovery process. The V$FAST_START_SERVERS columns and their definitions are:

State	The value of state is either "IDLE" or "RECOVERING".
Undoblocksdone	The percentage of the assigned work completed by the server.
PID	The oracle PID.

V$FAST_START_TRANSACTIONS View

This view provides information about recovery progress for each rollback transaction. This view has one row for each rollback transaction. The V$FAST_START_TRANSACTIONS columns and their definitions are:

Rollback segment number	The undo segment number of the transaction.
Slot	The slot the transaction occupies within the rollback segment.
Wrap	The incarnation number of the slot.
State	The state of the transaction may be one of "TO BE RECOVERED", "RECOVERING", or "RECOVERED".
Work_done	The percentage of recovery completed on this transaction.
Oracle PID	The ID of the current server process that recovery of this transaction has been assigned to.
Time elapsed	The amount of time in seconds that has elapsed since recovery on the transaction began.
Parent xid	The transaction ID of the parent transaction. Valid only for PDML transactions.

Oracle updates both tables continuously during Fast-start Parallel Recovery.

Determining a Degree of Parallelism for Fast-start Parallel Recovery

Tune the parameter FAST_START_PARALLEL_ROLLBACK to balance the performance of recovery and system resource use. Heavy Fast-start Parallel Recovery processing can degrade response times of unrelated database operations due to CPU and disk use. If you need to adjust the processing of Fast-start Parallel Recovery, change the value for FAST_START_PARALLEL_ROLLBACK at any time, even while the server is running. When you change this value during recovery, SMON restarts recovery with the new degree of parallelism.

Change the value of FAST_START_PARALLEL_ROLLBACK to HIGH when you have more recovery work to do. Set it to FALSE when you do not want parallel recovery to occupy the CPU. The CPU_COUNT parameter is correctly set because Oracle spawns 2 or 4 times the number of recovery servers.

Part III

Application Design Tools for Designers and DBAs

Part III discusses how to tune your database and the various methods you use to access data for optimal database performance. The chapters in Part 3 are:

12
Overview of Diagnostic Tools

This chapter introduces the full range of diagnostic tools for monitoring production systems and determining performance problems.

Topics in this chapter include:

- Sources of Data for Tuning
- Dynamic Performance Views
- Oracle and SNMP Support
- EXPLAIN PLAN
- Oracle Trace and Oracle Trace Data Viewer
- The SQL Trace Facility and TKPROF
- Supported Scripts
- Application Registration
- Oracle Enterprise Manager, Packs, and Applications
- Oracle Parallel Server Management
- Tools You May Have Developed

Sources of Data for Tuning

This section describes the various sources of data for tuning. Many of these sources may be transient. They include:

- Data Volumes
- Online Data Dictionary

- Operating System Tools
- Dynamic Performance Tables
- Oracle Trace and Oracle Trace Data Viewer
- SQL Trace Facility
- Alert Log
- Application Program Output
- Users
- Initialization Parameter Files
- Program Text
- Design (Analysis) Dictionary
- Comparative Data

Data Volumes

The tuning data source most often overlooked is the data itself. The data may contain information about how many transactions were performed and at what time. The number of rows added to an audit table, for example, can be the best measure of the amount of useful work done; this is known as "the throughput". Where such rows contain a timestamp, you can query the table and use a graphics package to plot throughput against dates and times. D ate- and time-stamps need not be apparent to the rest of the application.

If your application does not contain an audit table, be cautious about adding one as it could hinder performance. Consider the trade-off between the value of obtaining the information and the performance cost of doing so.

Online Data Dictionary

The Oracle online data dictionary is a rich source of tuning data when used with the SQL statement ANALYZE. This statement stores cluster, table, column, and index statistics within the dictionary, primarily for use by the cost-based optimizer. The dictionary also defines the indexes available to help (or possibly hinder) performance.

Operating System Tools

Tools that gather data at the operating system level are primarily useful for determining scalability, but you should also consult them at an early stage in any tuning activity. In this way you can ensure that no part of the hardware platform is saturated. Network monitors are also required in distributed systems, primarily to check that no network resource is overcommitted. In addition, you can use a simple mechanism such as the UNIX ping command to establish message turnaround time.

See Also: Your operating system documentation for more information on platform-specific tools.

Dynamic Performance Tables

A number of V$ dynamic performance views are available to help you tune your system and investigate performance problems. They allow you access to memory structures within the SGA.

See Also: Chapter 16, "Dynamic Performance Views" and *Oracle8i Concepts* provides detailed information about each view.

Oracle Trace and Oracle Trace Data Viewer

Oracle Trace collects Oracle server event activity that includes all SQL and Wait events for specific database users. You can use this information to tune your databases and applications.

See Also: Oracle Trace and Wait events are described in more detail in Chapter 15, "Using Oracle Trace".

SQL Trace Facility

SQL trace files record SQL statements issued by a connected process and the resources used by these statements. In general, use V$ views to tune the instance and use SQL trace file output to tune the applications.

See Also: Chapter 14, "The SQL Trace Facility and TKPROF".

Alert Log

Whenever something unexpected happens in an Oracle environment, check the alert log to see if there is an entry at or around the time of the event.

Application Program Output

In some projects, all application processes (client-side) are instructed to record their own resource consumption to an audit trail. Where database calls are being made through a library, the response time of the client/server mechanism can be inexpensively recorded at the per-call level using an audit trail mechanism. Even without these levels of sophistication, which are not expensive to build or to run, simply preserving resource usages reported by a batch queue manager provides an excellent source of tuning data.

Users

Users normally provide a stream of information as they encounter performance problems.

Initialization Parameter Files

It is vital to have accurate data on exactly what the system was instructed to do and how it was to go about doing it. Some of this data is available from the Oracle parameter files.

Program Text

Data on what the application was to do is also available from the code of the programs or procedures where both the program logic and the SQL statements reside. Server-side code, such as stored procedures, constraints, and triggers, is in this context part of the same data population as client-side code. Tuners must frequently work in situations where the program source code is not available, either as a result of a temporary problem or because the application is a package for which the source code is not released. In such cases it is still important for the tuner to acquire program-to-object cross-reference information. For this reason executable code is a legitimate data source. Fortunately, SQL is held in text even in executable programs.

Design (Analysis) Dictionary

You can also use the design or analysis dictionary to track intended actions and resource use of the application. Only where the application has been entirely produced by code generators, however, can the design dictionary provide data that would otherwise have to be extracted from programs and procedures.

Comparative Data

Comparative data is invaluable in most tuning situations. Tuning is often conducted from a cold start at each site; the tuners arrive with whatever expertise and experience they may have, plus a few tools for extracting the data. Experienced tuners may recognize similarities in particular situations and attempt to apply a solution that worked elsewhere. Normally, such diagnoses are purely subjective.

Tuning is easier if baselines exist, such as capacity studies performed for this application or data from this or another site running the same application with acceptable performance. The task is then to modify the problematic environment to more closely resemble the optimized environments.

If no directly relevant data can be found, you can check data from similar platforms and similar applications to see if they have the same performance profile. There is no point in trying to tune out a particular effect if it turns out to be ubiquitous!

Dynamic Performance Views

A primary Oracle performance monitoring tool is the dynamic performance views Oracle provides to monitor your system. These views' names begin with "V$". This text demonstrates their use in performance tuning. The database user SYS owns these views, and administrators can grant any database user access to them. However, only some of these views are relevant to tuning your system.

See Also: Chapter 16, "Dynamic Performance Views" and *Oracle8i Concepts* provides detailed information about each view.

Oracle and SNMP Support

Simple Network Management Protocol (SNMP) enables users to write tools and applications. SNMP is acknowledged as the standard, open protocol for heterogeneous management applications. Oracle SNMP support enables Oracle databases to be discovered on the network and to be identified and monitored by SNMP-based management applications. Oracle supports several database management information bases (MIBs): the standard MIB for any database management system (independent of vendor), and Oracle-specific MIBs that contain Oracle-specific information. Some statistics mentioned in this manual are supported by these MIBs, and others are not. If you can obtain a statistic mentioned through SNMP, this fact is noted.

See Also: The *Oracle SNMP Support Reference Guide.*

EXPLAIN PLAN

EXPLAIN PLAN is a SQL statement listing the access path used by the query optimizer. Each plan output from the EXPLAIN PLAN command has a row that provides the statement type.

You should interpret EXPLAIN PLAN results with some discretion. Just because a plan does not seem efficient does not necessarily mean the statement runs slowly. Choose statements for tuning based on their actual resource consumption, not on a subjective view of their execution plans.

See Also: Chapter 13, "Using EXPLAIN PLAN" and the *Oracle8i SQL Reference*.

Oracle Trace and Oracle Trace Data Viewer

Oracle Trace collects significant Oracle server event data such as all SQL events and Wait events. SQL events include a complete breakdown of SQL statement activity, such as the parse, execute, and fetch operations. Data collected for server events includes resource usage metrics such as I/O and CPU consumed by a specific event.

Identifying resource-intensive SQL statements is easy with Oracle Trace. The Oracle Trace Data Viewer summarizes Oracle Trace data, including SQL statement metrics such as: average elapsed time, CPU consumption, and disk reads per rows processed.

Oracle Trace collections can be administered through Oracle Trace Manager. Oracle Trace Data Viewer and Oracle Trace Manager are available with the Oracle Diagnostics Pack.

See Also: Chapter 15, "Using Oracle Trace".

The SQL Trace Facility and TKPROF

The SQL trace facility can be enabled for any session. It records in an operating system text file the resource consumption of every parse, execute, fetch, commit, or rollback request made to the server by the session.

TKPROF summarizes the trace files produced by the SQL trace facility, optionally including the EXPLAIN PLAN output. TKPROF reports each statement executed with the resources it has consumed, the number of times it was called, and the number of rows it processed. It is thus quite easy to locate individual statements that are using the greatest amount of resources. With experience or with baselines available, you can gauge whether the resources used are reasonable.

Supported Scripts

Oracle provides many PL/SQL packages, thus a good number of SQL*Plus scripts supporting instance tuning are available. Examples include UTLBSTAT.SQL and UTLESTAT; SQLUTLCHAIN.SQL, UTLDTREE.SQL, and UTLLOCKT.SQL.

These statistical scripts support instance management, allowing you to develop performance history. You can use them to:

- Remove the need to issue DDL each time statistics are gathered.
- Separate data gathering from reporting and allow a range of observations to be taken at intervals during a period of representative system operation, and then to allow the statistics to be reported from any start point to any end point.
- Report a number of indicative ratios that you can use to determine whether the instance is adequately tuned.
- Present LRU statistics from the buffer cache in a usable form.

Application Registration

You can register the name of an application with the database and actions performed by that application. Registering the application allows system administrators and tuners to track performance by module. System administrators can also use this information to track resource usage by module. When an application registers with the database, its name and actions are recorded in the V$SESSION and V$SQLAREA views.

In addition, Oracle Trace can collect application registration data.

> **See Also:** The *Oracle Trace User's Guide*, the *Oracle Trace Developer's Guide* for additional information, and Chapter 5, "Registering Applications".

Oracle Enterprise Manager, Packs, and Applications

This section describes Oracle Enterprise Manager, its packs, and several of its most useful diagnostic and tuning applications. It covers:

- Introduction to Oracle Enterprise Manager
- Oracle Diagnostics Pack
    - Oracle Capacity Planner

    - Oracle Performance Manager
    - Oracle Advanced Event Tests
    - Oracle Trace
- Oracle Tuning Pack
    - Oracle Expert
    - Oracle Index Tuning Wizard
    - Oracle SQL Analyze
    - Oracle Auto-Analyze
    - Oracle Tablespace Manager

Introduction to Oracle Enterprise Manager

Oracle is addressing the need for a sophisticated, database, systems-management environment with the Oracle Enterprise Manager platform. This tool provides comprehensive management for Oracle environments.

You can use Oracle Enterprise Manager to manage the wide range of Oracle implementations: departmental to enterprise, replication configurations, web servers, media servers, and so forth. Oracle Enterprise Manager includes:

- A centralized console from which you can run administrative tasks and applications.
- Support to run the Oracle Enterprise Manager console and database administration applications from within a web browser.
- A lightweight, 3-tier architecture offering unparalleled scalability and failover capability, assuring constant availability of critical management services.
- A centralized repository storing management data for any given environment. Oracle Enterprise Manager supports teams of administrators responsible for cooperatively managing distributed systems.
- Common services for event management, service discovery, and job creation and control.
- Server-side, intelligent agent for remote monitoring of events, running jobs, and communicating with the management console.
- Low overhead framework for collecting and managing real-time and historical performance data.

- Applications for administering Oracle databases for security, storage, backup, recovery, import, and software distribution.
- Layered applications for managing replication, Oracle Parallel Server, and other Oracle server configurations.
- Optional products for monitoring, diagnosing, and planning, known as Oracle Diagnostics Pack.
- Optional products for tuning applications, databases, and systems, known as Oracle Tuning Pack.
- Optional products for managing Oracle metadata changes, known as Oracle Change Management Pack.

The Oracle Enterprise Manager packs provide a set of windows-based and java-based applications built on the Oracle Enterprise Manager systems management technology. Due to the nature of this manual, the Oracle Change Management Pack will not be presented.

Oracle Diagnostics Pack

The Oracle Diagnostics Pack monitors, diagnoses, and maintains the health of databases, operating systems, and applications. Both historical and real-time analysis are used to automatically avoid problems before they occur. The pack provides powerful capacity planning features enabling users to easily plan and track future system resource requirements.

Oracle Diagnostics Pack components include: Oracle Capacity Planner, Oracle Performance Manager, Oracle Advanced Event Tests, and Oracle Trace. The following sections describe each component.

Oracle Capacity Planner

Use the Oracle Capacity Planner to collect and analyze historical performance data for your Oracle database and operating system. Oracle Capacity Planner allows you to specify the performance data you want to collect, collection intervals, load schedules, and data management policies. You can also use Oracle Capacity Planner's in-depth analyses and reports to explore the collected data, to format it into easy-to-use graphs and reports, and to analyze it to predict future resource needs.

Oracle Performance Manager

Oracle Performance Manager captures, computes, and presents performance data for your database and operating system, allowing you to monitor key metrics required to effectively use memory, minimize disk I/O, and to avoid resource contention. It provides a graphical, real-time view of the performance metrics and lets you drill down into a monitoring view for quick access to detailed data for performance problem solving. The performance data is captured and displayed in real-time mode. You can also record the data for replay.

Oracle Performance Manager includes a large set of predefined charts. You can also create your own charts. The graphical monitor is customizable and extensible. You can display monitored information in a variety of two- or three-dimensional graphical views, such as tables, line, bar, cube, and pie charts. You can also customize the monitoring rate.

In addition, Oracle Performance Manager provides a focused view of database activity by database session. The Top Sessions chart extracts and analyzes sample dynamic Oracle performance data by session, automatically determining the top Oracle users based on a specific selection criteria, such as memory consumption, CPU usage, or file I/O activity.

Also, the Database Locks chart within Oracle Performance Manager displays database locks, including details such as the locking user, lock type, object locked, and mode held and requested.

Oracle Advanced Event Tests

Oracle Diagnostics Pack includes Oracle Advanced Event Tests. This is a set of agent-monitored host and database events running on the Oracle Event Management System. You can launch advanced event tests from the console to automatically detect problems on managed servers. Oracle Advanced Event Tests includes predefined events for monitoring database services and system events affecting database performance.

For example, performance-monitoring events include I/O monitoring, memory-structure performance, and user program-response time. I/O monitoring covers disk I/O rates and SQL*Net I/O rates. The tool even allows you to specify an I/O rate threshold; you will receive a warning when this threshold is exceeded.

Memory-structure performance monitoring covers hit rates for the library cache, data dictionary, and database buffers. In addition, you also have the flexibility of monitoring any statistic captured by the dynamic performance table, V$SYSSTAT.

You can use Oracle Advanced Event Tests to monitor the status and performance of Oracle storage structures and to detect problems with excessive CPU utilization, excessive CPU load or paging, and disk capacity problems.

In addition to alerting an administrator, Oracle Advanced Event Tests also can be configured to automatically correct the problem event. Using a "Fixit Job", a predetermined action will automatically occur when an event-alert level is reached.

Oracle Tuning Pack

Oracle Tuning Pack optimizes system performance by identifying and tuning major database and application bottlenecks such as inefficient SQL, poor data structures, and improper use of system resources. The pack proactively discovers tuning opportunities and automatically generates the analysis and required changes to tune the system. Inherent in the product are powerful teaching tools that train DBAs how to tune as they work.

Oracle Expert

Oracle Expert provides automated database performance tuning. Performance problems detected by Oracle Diagnostics Pack and other Oracle monitoring applications can be analyzed and solved with Oracle Expert. Oracle Expert automates the process of collecting and analyzing data and contains a rules-based inference engine that provides "expert" database tuning recommendations, implementation scripts, and reports.

Oracle SQL Analyze

Oracle SQL Analyze identifies and helps you tune problematic SQL statements. Use SQL Analyze to detect resource-intensive SQL statements, examine a SQL statement’s execution plan, benchmark and compare various optimizer modes and versions of the statement, and to generate alternative SQL to improve application performance.

Oracle Tablespace Manager

You can use Oracle Tablespace Manager to identify and correct Oracle space management problems. Oracle Tablespace Manager has three major features: a Tablespace Allocation graphic, a Tablespace Reorganization tool, and a Tablespace Analyzer tool.

The Tablespace Allocation graphic on the Segments and Extents Information page provides a complete picture of the characteristics of all tablespaces associated with a particular Oracle instance, including: tablespace datafiles and segments, total data blocks, free data blocks, and percentage of free blocks available in the tablespace's current storage allocation.

Use the Reorganization tool to rebuild specific objects or an entire tablespace for improved space usage and increased performance. Use the Analyzer tool to automatically keep database statistics up-to-date.

Oracle Index Tuning Wizard

The Oracle Index Tuning Wizard automatically identifies tables that would benefit from index changes, determines the best index strategy for each table, presents its findings for verification, and allows you to implement its recommendations.

Oracle Auto-Analyze

Use Oracle Auto-Analyze to maintain your Oracle database statistics. Auto-Analyze runs during a user-specified database maintenance period, thereby reducing adverse performance effects of updating stale statistics. During this maintenance period, Auto-Analyze checks specific schemas for objects that require updating. It also prioritizes the order of objects that require updating and updates the statistics. If the statistics update does not complete during the maintenance period, Auto-Analyze maintains the state of the update operation and resumes updating during the next maintenance period.

Oracle Parallel Server Management

Oracle Parallel Server Management (OPSM) is a comprehensive and integrated system management solution for the Oracle Parallel Server. Use OPSM to manage multi-instance databases running in heterogeneous environments through an open client-server architecture.

In addition to managing parallel databases, you can use OPSM to schedule jobs, perform event management, monitor performance, and obtain statistics to tune parallel databases.

For more information about OPSM, please refer to the *Oracle Parallel Server Management Configuration Guide for UNIX* and the *Oracle Parallel Server Management User's Guide*. For installation instructions, refer to your platform-specific installation guide.

Tools You May Have Developed

At some sites, DBAs have designed in-house performance tools. Such tools might include:

- Free space monitors to determine whether tables have enough space to extend
- Lock monitoring tools
- Schema description scripts to show tables and their associated indexes
- Tools to show default and temporary tablespaces per user

You can integrate such programs with Oracle by setting them to run automatically.

13 Using EXPLAIN PLAN

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

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

Introduction to EXPLAIN PLAN

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

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

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

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

> **See Also:** Chapter 14, "The SQL Trace Facility and TKPROF" and Chapter 15, "Using Oracle Trace".

Creating the Output Table

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

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

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

```
CREATE TABLE plan_table
(statement_id     VARCHAR2(30),
 timestamp        DATE,
 remarks          VARCHAR2(80),
 operation        VARCHAR2(30),
 options          VARCHAR2(30),
 object_node      VARCHAR2(128),
 object_owner     VARCHAR2(30),
 object_name      VARCHAR2(30),
 object_instance  NUMERIC,
 object_type      VARCHAR2(30),
 optimizer        VARCHAR2(255),
 search_columns   NUMERIC,
 id              NUMERIC,
 parent_id       NUMERIC,
 position        NUMERIC,
 cost            NUMERIC,
 cardinality     NUMERIC,
 bytes           NUMERIC,
 other_tag       VARCHAR2(255)
 other           LONG);
```

Displaying PLAN_TABLE Output

Display the most recent plan table output using the scripts:

- UTLPLS.SQL - To show plan table output for serial processing.
- UTLXPLP.SQL - To show plan table output with parallel execution columns.

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

See Also: "Selecting PLAN_TABLE Output in Nested Format" on page 13-27.

Output Table Columns

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

Table 13–1 PLAN_TABLE Columns

Column	Description
STATEMENT_ID	The value of the optional STATEMENT_ID parameter specified in the EXPLAIN PLAN statement.
TIMESTAMP	The date and time when the EXPLAIN PLAN statement was issued.
REMARKS	Any comment (of up to 80 bytes) you wish to associate with each step of the explained plan. If you need to add or change a remark on any row of the PLAN_TABLE, use the UPDATE statement to modify the rows of the PLAN_TABLE.

Table 13–1 PLAN_TABLE Columns

OPERATION	The name of the internal operation performed in this step. In the first row generated for a statement, the column contains one of the following values:
	DELETE STATEMENT
	INSERT STATEMENT
	SELECT STATEMENT
	UPDATE STATEMENT
	See Table 13–4 for more information on values for this column.

OPTIONS	A variation on the operation described in the OPERATION column. See Table 13–4 for more information on values for this column.
OBJECT_NODE	The name of the database link used to reference the object (a table name or view name). For local queries using parallel execution, this column describes the order in which output from operations is consumed.
OBJECT_OWNER	The name of the user who owns the schema containing the table or index.
OBJECT_NAME	The name of the table or index.
OBJECT_INSTANCE	A number corresponding to the ordinal position of the object as it appears in the original statement. The numbering proceeds from left to right, outer to inner with respect to the original statement text. View expansion will result in unpredictable numbers.
OBJECT_TYPE	A modifier that provides descriptive information about the object; for example, NON-UNIQUE for indexes.
OPTIMIZER	The current mode of the optimizer.
SEARCH_COLUMNS	Not currently used.
ID	A number assigned to each step in the execution plan.
PARENT_ID	The ID of the next execution step that operates on the output of the ID step.

Table 13–1 PLAN_TABLE Columns

POSITION	The order of processing for steps that all have the same PARENT_ID.
OTHER	Other information that is specific to the execution step that a user may find useful.
OTHER_TAG	Describes the contents of the OTHER column. See Table 13–2 for more information on the possible values for this column.
DISTRIBUTION	Stores the method used to distribute rows from "producer" query servers to "consumer" query servers. For more information about consumer and producer query servers, please see *Oracle8i Concepts*.
Pstart	The start partition of a range of accessed partitions. It can take one of the following values: *n* indicates that the start partition has been identified by the SQL compiler and its partition number is given by *n*. KEY indicates that the start partition will be identified at execution time from partitioning key values. ROW LOCATION indicates that the start partition (same as the stop partition) will be computed at execution time from the location of each record being retrieved. The record location is obtained by a user or from a global index. INVALID indicates that the range of accessed partitions is empty.

Table 13–1 PLAN_TABLE Columns

Pstop	The stop partition of a range of accessed partitions. It can take one of the following values: *n* indicates that the stop partition has been identified by the SQL compiler and its partition number is given by *n*. KEY indicates that the stop partition will be identified at execution time from partitioning key values. ROW LOCATION indicates that the stop partition (same as the start partition) will be computed at execution time from the location of each record being retrieved. The record location is obtained by a user or from a global index. INVALID indicates that the range of accessed partitions is empty.
PID	The step that has computed the pair of values of the Pstart and Pstop columns.
COST	The cost of the operation as estimated by the optimizer's cost-based approach. For statements that use the rule-based approach, this column is null. Cost is not determined for table access operations. The value of this column does not have any particular unit of measurement, it is merely a weighted value used to compare costs of execution plans.
CARDINALITY	The estimate by the cost-based approach of the number of rows accessed by the operation.
BYTES	The estimate by the cost-based approach of the number of bytes accessed by the operation.

Table 13–2 describes the values that may appear in the OTHER_TAG column.

Table 13–2 Values of OTHER_TAG Column of the PLAN_TABLE

OTHER_TAG Text (examples)	Meaning	Interpretation
blank		Serial execution.
serial_from_remote (S -> R)	Serial from remote	Serial execution at a remote site.
serial_to_parallel (S -> P)	Serial to parallel	Serial execution; output of step is partitioned or broadcast to parallel execution servers.
parallel_to_parallel (P - > P)	Parallel to parallel	Parallel execution; output of step is repartitioned to second set of parallel execution servers.
parallel_to_serial (P -> S)	Parallel to serial	Parallel execution; output of step is returned to serial "query coordinator" process.
parallel_combined_with_parent (PwP)	Parallel combined with parent	Parallel execution; output of step goes to next step in same parallel process. No interprocess communication to parent.
parallel_combined_with_child (PwC)	Parallel combined with child	Parallel execution; input of step comes from prior step in same parallel process. No interprocess communication from child.

Table 13–3 describes the values that can appear in the DISTRIBUTION column:

Table 13–3 Values of DISTRIBUTION Column of the PLAN_TABLE

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

Table 13–4 lists each combination of OPERATION and OPTION produced by the EXPLAIN PLAN statement and its meaning within an execution plan.

Table 13–4 OPERATION and OPTION 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.
	CONVERSION	TO ROWIDS converts bitmap representations to actual rowids that can be used to access the table. FROM ROWIDS converts the rowids to a bitmap representation. COUNT returns the number of rowids if the actual values are not needed.
	INDEX	SINGLE VALUE looks up the bitmap for a single key value in the index. RANGE SCAN retrieves bitmaps for a key value range. FULL SCAN: Performs a full scan of a bitmap index if there is no start or stop key.
	MERGE	Merges several bitmaps resulting from a range scan into one bitmap.
	MINUS	Subtracts bits of one bitmap from another. Row source is used for negated predicates. Can be used only if there are nonnegated predicates yielding a bitmap from which the subtraction can take place. An example appears in "Bitmap Indexes and EXPLAIN PLAN".
	OR	Computes the bitwise OR of two bitmaps.
CONNECT BY		Retrieves rows in hierarchical order for a query containing a CONNECT BY clause.
CONCATENATION		Operation accepting multiple sets of rows returning the union-all of the sets.
COUNT		Operation counting the number of rows selected from a table.
	STOPKEY	Count operation where the number of rows returned is limited by the ROWNUM expression in the WHERE clause.

Table 13–4 OPERATION and OPTION Values Produced by EXPLAIN PLAN

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

Table 13–4 OPERATION and OPTION Values Produced by EXPLAIN PLAN

OPERATION	OPTION	Description
NESTED LOOPS (These are join operations.)		Operation accepting two sets of rows, an outer set and an inner set. Oracle compares each row of the outer set with each row of the inner set, returning rows that satisfy a condition.
	OUTER	Nested loops operation to perform an outer join statement.
PARTITION	SINGLE	Access one partition.
	ITERATOR	Access many partitions (a subset).
	ALL	Access all partitions.
	INLIST	Similar to iterator but based on an inlist predicate.
	INVALID	Indicates that the partition set to be accessed is empty.
		Iterates over the operation below it, for each partition in the range given by the PARTITION_START and PARTITION_STOP columns. PARTITION describes partition boundaries applicable to a single partitioned object (table or index) or to a set of equi-partitioned objects (a partitioned table and its local indexes). The partition boundaries are provided by the values of pstart and pstop of the PARTITION. Refer to Table 13–1 for valid values of partition start/stop.
PROJECTION		Internal operation.
REMOTE		Retrieval of data from a remote database.
SEQUENCE		Operation involving accessing values of a sequence.
SORT	AGGREGATE	Retrieval of a single row that is the result of applying a group function to a group of selected rows.
	UNIQUE	Operation sorting a set of rows to eliminate duplicates.
	GROUP BY	Operation sorting a set of rows into groups for a query with a GROUP BY clause.
	JOIN	Operation sorting a set of rows before a merge-join.
	ORDER BY	Operation sorting a set of rows for a query with an ORDER BY clause.

Table 13–4 OPERATION and OPTION Values Produced by EXPLAIN PLAN

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

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

Bitmap Indexes and EXPLAIN PLAN

Index row sources appear in the EXPLAIN PLAN output with the word BITMAP indicating the type. Consider the following sample query and plan, in which the TO ROWIDS option is used to generate the ROWIDs that are necessary for table access.

```
EXPLAIN PLAN FOR
 SELECT * FROM T
 WHERE
 C1 = 2 AND C2 <> 6
 OR
 C3 BETWEEN 10 AND 20;

SELECT STATEMENT
 TABLE ACCESS  T  BY ROWID
 BITMAP CONVERSION TO ROWIDS
 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.

EXPLAIN PLAN and Partitioned Objects

Use EXPLAIN PLAN to see how Oracle will access 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 How EXPLAIN PLAN Displays Range and Hash Partitioning

Consider the following table, EMP_RANGE, partitioned by range on HIREDATE to illustrate how pruning is displayed. Assume that the tables EMP and DEPT from a standard Oracle schema exist.

```
CREATE TABLE EMP_RANGE
 PARTITION BY RANGE(HIREDATE)
 (
 PARTITION EMP_P1 VALUES LESS THAN (TO_DATE('1-JAN-1981','DD-MON-YYYY')),
 PARTITION EMP_P2 VALUES LESS THAN (TO_DATE('1-JAN-1983','DD-MON-YYYY')),
 PARTITION EMP_P3 VALUES LESS THAN (TO_DATE('1-JAN-1985','DD-MON-YYYY')),
 PARTITION EMP_P4 VALUES LESS THAN (TO_DATE('1-JAN-1987','DD-MON-YYYY')),
 PARTITION EMP_P5 VALUES LESS THAN (TO_DATE('1-JAN-1989','DD-MON-YYYY'))
 )
 AS SELECT * FROM EMP;
```

Example 1:

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

Then enter the following to display the EXPLAIN PLAN output:

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

Oracle displays something similar to:

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

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

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

Example 2:

```
   EXPLAIN PLAN FOR SELECT * FROM EMP_RANGE
   WHERE HIREDATE >= TO_DATE('1-JAN-1985','DD-MON-YYYY');

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

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

Example 3:

```
   EXPLAIN PLAN FOR SELECT * FROM EMP_RANGE
   WHERE HIREDATE < TO_DATE('1-JAN-1981','DD-MON-YYYY');

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

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

Plans for Hash Partitioning

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

Pruning Information with Composite Partitioned Objects

To illustrate how Oracle displays pruning information for composite partitioned objects, consider the table EMP_COMP that is range partitioned on HIREDATE and subpartitioned by hash on DEPTNO.

```
CREATE TABLE EMP_COMP PARTITION BY RANGE(HIREDATE) SUBPARTITION BY HASH(DEPTNO)
 SUBPARTITIONS 3
 (
 PARTITION EMP_P1 VALUES LESS THAN (TO_DATE('1-JAN-1981','DD-MON-YYYY')),
 PARTITION EMP_P2 VALUES LESS THAN (TO_DATE('1-JAN-1983','DD-MON-YYYY')),
 PARTITION EMP_P3 VALUES LESS THAN (TO_DATE('1-JAN-1985','DD-MON-YYYY')),
 PARTITION EMP_P4 VALUES LESS THAN (TO_DATE('1-JAN-1987','DD-MON-YYYY')),
 PARTITION EMP_P5 VALUES LESS THAN (TO_DATE('1-JAN-1989','DD-MON-YYYY'))
  )
 AS SELECT * FROM EMP;
```

Example 1:

```
   EXPLAIN PLAN FOR SELECT * FROM EMP_COMP;

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

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

Example 2:

```
EXPLAIN PLAN FOR SELECT * FROM EMP_COMP WHERE HIREDATE =
TO_DATE('15-FEB-1987', 'DD-MON-YYYY');
```

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

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

Example 3:

```
EXPLAIN PLAN FOR SELECT * FROM EMP_COMP WHERE DEPTNO = 20;
```

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

In this 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.

Example 4:

```
   VARIABLE DNO NUMBER;
   EXPLAIN PLAN FOR SELECT * FROM EMP_COMP WHERE DEPTNO = :DNO;
```

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

Example 4 is the same as example 3 except that "DEPTNO = 20" has been replaced by "DEPTNO = :DNO". In this case, the subpartition number is unknown at compile time and a hash partition row source is allocated. The option is SINGLE for that row source because Oracle accesses only one subpartition within each partition. The PARTITION START and PARTITION STOP is set to "KEY". This means Oracle will determine the number of the subpartition at run time.

Partial Partition-wise Joins

Example 1:

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

```
   ALTER TABLE EMP PARALLEL 2;
      STATEMENT PROCESSED.
   ALTER TABLE DEPT PARALLEL 2;
      STATEMENT PROCESSED.
```

To show the plan for the query, enter:

```
   EXPLAIN PLAN FOR SELECT /*+ ORDERED USE_HASH(D) */ ENAME, DNAME
     FROM EMP_RANGE E, DEPT D
     WHERE E.DEPTNO = D.DEPTNO
     AND E.HIREDATE > TO_DATE('29-JUN-1986','DD-MON-YYYY');
```

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

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

Example 2:

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

```
ALTER TABLE EMP_COMP PARALLEL 2;
  STATEMENT PROCESSED.
EXPLAIN PLAN FOR SELECT /*+ ORDERED USE_HASH(D) */ ENAME, DNAME
  FROM EMP_COMP E, DEPT D
  WHERE E.DEPTNO = D.DEPTNO
  AND E.HIREDATE > TO_DATE('13-MAR-1985','DD-MON-YYYY');
```

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

Full Partition-wise Joins

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

To create the table DEPT_HASH, enter:

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

To show the plan for the query, enter:

```
EXPLAIN PLAN FOR SELECT /*+ ORDERED USE_HASH(D) */ ENAME, DNAME
  FROM EMP_COMP E, DEPT_HASH D
  WHERE E.DEPTNO = D.DEPTNO
  AND E.HIREDATE > TO_DATE('29-JUN-1986','DD-MON-YYYY');
```

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

INLIST ITERATOR and EXPLAIN PLAN

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

```
SELECT * FROM EMP WHERE EMPNO IN (7876, 7900, 7902);
```

The EXPLAIN PLAN output appears as follows:

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

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

Index Column

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

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

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

Index and Partition Column

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

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

Partition Column

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

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

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

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

DOMAIN INDEX 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
```

Formatting EXPLAIN PLAN Output

This section shows options for formatting EXPLAIN PLAN output

- Using the EXPLAIN PLAN Statement
- Selecting PLAN_TABLE Output in Table Format
- Selecting PLAN_TABLE Output in Nested Format

Note: The output of the EXPLAIN PLAN statement reflects the behavior of the Oracle optimizer. As the optimizer evolves between releases of the Oracle server, output from the EXPLAIN PLAN statement is also likely to evolve.

Using the EXPLAIN PLAN Statement

The following example shows a SQL statement and its corresponding execution plan generated by EXPLAIN PLAN. The sample query retrieves names and related information for employees whose salary is not within any range of the SALGRADE table:

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

This EXPLAIN PLAN statement generates an execution plan and places the output in PLAN_TABLE:

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

Selecting PLAN_TABLE Output in Table Format

This SELECT statement:

```
SELECT operation, options, object_name, id, parent_id, position, cost, cardinality,
other_tag, optimizer
   FROM plan_table
   WHERE statement_id = ’Emp_Sal’
   ORDER BY id;
```

Generates this output:

```
  OPERATION  OPTIONS OBJECT_NAME ID PARENT_ID POSITION COST CARDINALITY BYTES OTHER_TAG
OPTIMIZER

-----------------------------------------------------------------------------------------
  SELECT STATEMENT                  0             2    2           1    62
CHOOSE
  FILTER                            1         0    1
  NESTED LOOPS                      2         1    1    2           1    62
  TABLE ACCESS FULL    EMP          3         2    1    1           1    40
ANALYZED
  TABLE ACCESS FULL    DEPT         4         2    2                4    88
ANALYZED
  TABLE ACCESS FULL    SALGRADE     5         1    2    1           1    13
ANALYZED
```

The ORDER BY clause returns the steps of the execution plan sequentially by ID value. However, Oracle does not perform the steps in this order. PARENT_ID receives information from ID, yet more than one ID step fed into PARENT_ID.

For example, step 2, a merge join, and step 6, a table access, both fed into step 1. A nested, visual representation of the processing sequence is shown in the next section.

The value of the POSITION column for the first row of output indicates the optimizer’s estimated cost of executing the statement with this plan to be 5. For the other rows, it indicates the position relative to the other children of the same parent.

Note: A CONNECT BY does not preserve ordering. To have rows come out in the correct order in this example, you must either truncate the table first, or else create a view and select from the view. For example:

```
CREATE VIEW test AS
SELECT id, parent_id,
lpad(’ ’, 2*(level-1))||operation||’ ’||options||’ ’||object_name||’ ’||
       decode(id, 0, ’Cost = ’||position) "Query Plan"
FROM plan_table
START WITH id = 0 and statement_id = 'TST'
CONNECT BY prior id = parent_id and statement_id = ’TST’;
SELECT * FROM foo ORDER BY id, parent_id;
```

This yields results as follows:

```
 ID  PAR Query Plan
 --- --- --------------------------------------------------
  0     Select Statement   Cost = 69602
  1   0   Nested Loops
  2   1     Nested Loops
  3   2       Merge Join
  4   3         Sort Join
  5   4           Table Access Full T3
  6   3         Sort Join
  7   6           Table Access Full T4
  8   2       Index Unique Scan T2
  9   1     Table Access Full T1
10 rows selected.
```

Selecting PLAN_TABLE Output in Nested Format

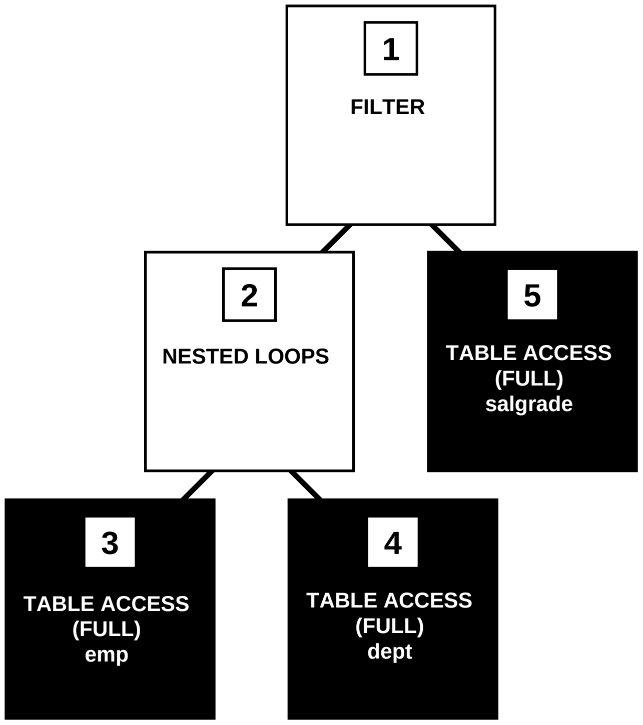
This type of SELECT statement generates a nested representation of the output that more closely depicts the processing order used for the SQL statement.

```
SELECT LPAD(' ',2*(LEVEL-1))||operation||' '||options
  ||' '||object_name
  ||' '||DECODE(id, 0, 'Cost = '||position) "Query Plan"
  FROM plan_table
  START WITH id = 0 AND statement_id = 'Emp_Sal'
  CONNECT BY PRIOR id = parent_id AND statement_id ='Emp_Sal';

Query Plan
------------------------------
SELECT STATEMENT   Cost = 5
  FILTER
     NESTED LOOPS
        TABLE ACCESS FULL EMP
        TABLE ACCESS FULL DEPT
     TABLE ACCESS FULL SALGRADE
```

The order resembles a tree structure, as illustrated in Figure 13–1.

Figure 13–1 Tree Structure of an Execution Plan



Tree structures illustrate how SQL statement execution operations feed one another. Oracle assigns each step in the execution plan a number representing the ID column of the PLAN_TABLE. Each step is depicted by a "node". The result of each node's operation passes to its parent node, which uses it as input.

EXPLAIN PLAN Restrictions

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

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

See Also: Chapter 14, "The SQL Trace Facility and TKPROF".

14

The SQL Trace Facility and TKPROF

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

Introduction to SQL Trace and TKPROF

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

About the SQL Trace Facility

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

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

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

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

About TKPROF

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

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

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

Using the SQL Trace Facility and TKPROF

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

1. Set initialization parameters for trace file management.
2. Enable the SQL trace facility for the desired session and run your application. This step produces a trace file containing statistics for the SQL statements issued by the application.
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 stores the statistics in the database.
4. Interpret the output file created in Step 3.
5. Optionally, run the SQL script produced in Step 3 to store the statistics in the database.

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

Step 1: Set Initialization Parameters for Trace File Management

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

Before enabling the SQL trace facility, you should:

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

Table 14–1 SQL Trace Facility Dynamic Initialization Parameters

Parameter	Notes
TIMED_STATISTICS	This parameter 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 session parameter.
MAX_DUMP_FILE_SIZE	When the SQL trace facility is enabled at the instance level, every call to the server produces a text line in a file in your operating system's file format. The maximum size of these files (in operating system blocks) is limited by the initialization parameter MAX_DUMP_FILE_SIZE. The default is 500. If you find that your trace output is truncated, increase the value of this parameter before generating another trace file. This is a session parameter.
USER_DUMP_DEST	This parameter must specify fully the destination for the trace file according to the conventions of your operating system. The default value for this parameter is the default destination for system dumps on your operating system.This value can be modified with ALTER SYSTEM SET USER_DUMP_DEST=*newdir*. This is a system parameter.

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

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

3. If your operating system retains multiple versions of files, be sure your version limit is high enough to accommodate the number of trace files you expect the SQL trace facility to generate.

4. The generated trace files may be owned by an operating system user other than yourself. This user must make the trace files available to you before you can use TKPROF to format them.

Step 2: Enable the SQL Trace Facility

You can enable the SQL trace facility for a session or for the instance. This section covers:

- Enabling the SQL Trace Facility for Your Current Session

- Enabling the SQL Trace Facility for an Instance

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

Enabling the SQL Trace Facility for Your Current Session

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

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

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

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

```
ALTER SESSION SET SQL_TRACE = FALSE;
```

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

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

Enabling the SQL Trace Facility for an Instance

To enable the SQL trace facility for your instance, set the value of the SQL_TRACE initialization parameter to TRUE. Statistics will be collected for all sessions.

Once the SQL trace facility has been enabled for the instance, you can disable it for an individual session by entering:

```
ALTER SESSION SET SQL_TRACE = FALSE;
```

Step 3: Format Trace Files with TKPROF

This section covers:

- Sample TKPROF Output
- Syntax of TKPROF
- TKPROF Statement Examples

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

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

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

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

Sample TKPROF Output

Sample output from TKPROF is as follows:

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

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

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

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

For this statement, TKPROF output includes the following information:

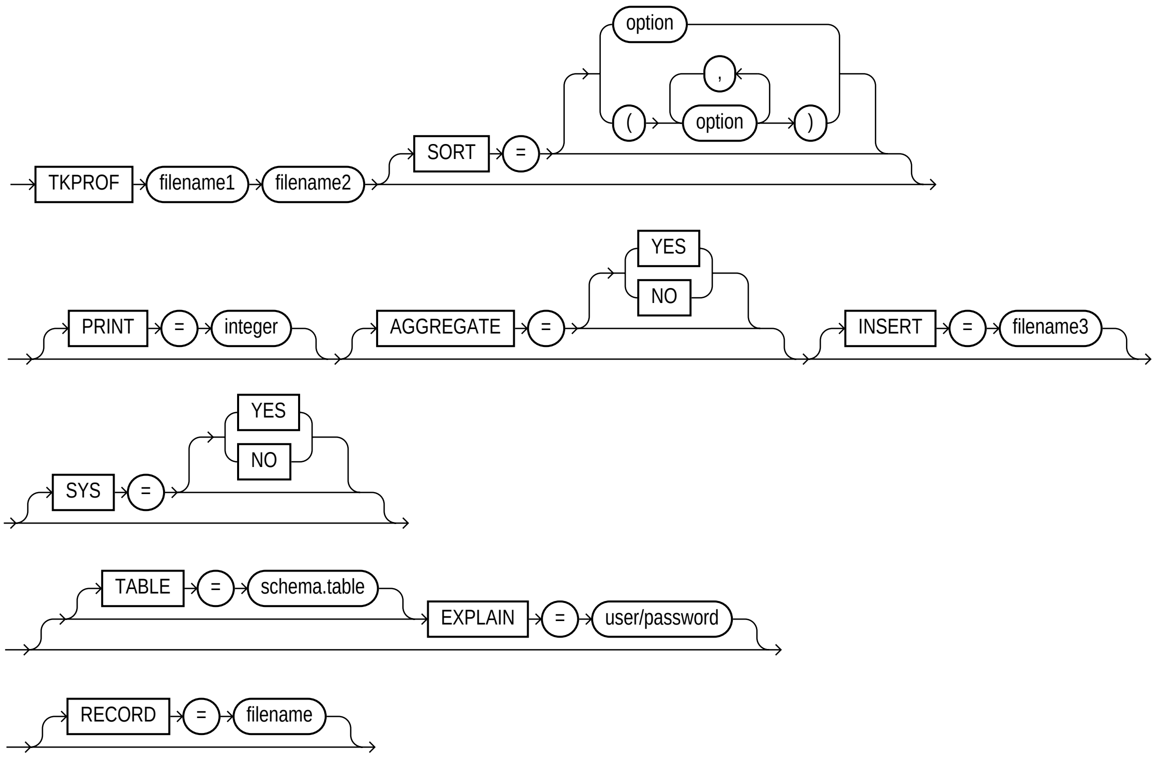
- The text of the SQL statement

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

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

Syntax of TKPROF

Invoke TKPROF using this syntax:



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

Use the following arguments with TKPROF:

Table 14–2 TKPROF Arguments

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

Table 14–2 TKPROF Arguments

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

Table 14–2 TKPROF Arguments

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

TKPROF Statement Examples

This section provides two brief examples of TKPROF usage. For an complete example of TKPROF output, see "TKPROF Output Example" on page 14-24.

Example 1

If you are processing a large trace file using a combination of SORT parameters and the PRINT parameter, you can produce a TKPROF output file containing only the highest resource-intensive statements. For example, the following statement prints the ten statements in the trace file that have generated the most physical I/O:

```
TKPROF ora53269.trc ora 53269.prf
SORT = (PRSDSK, EXEDSK, FCHDSK)
PRINT = 10
```

Example 2

This example runs TKPROF, accepts a trace file named "dlsun12_jane_fg_svrmgr_007.trc", and writes a formatted output file named "outputa.prf":

```
TKPROF DLSUN12_JANE_FG_SVRMGR_007.TRC OUTPUTA.PRF
EXPLAIN=SCOTT/TIGER TABLE=SCOTT.TEMP_PLAN_TABLE_A INSERT=STOREA.SQL SYS=NO
SORT=(EXECPU,FCHCPU)
```

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

Note the other parameters in this example:

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

Step 4: Interpret TKPROF Output

This section provides pointers for interpreting TKPROF output.

- Tabular Statistics
- Library Cache Misses
- Statement Truncation
- User Issuing the SQL Statement
- Execution Plan
- Deciding Which Statements to Tune

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

See Also: *Oracle8i Reference* for a description of statistics in V$SYSSTAT and V$SESSTAT.

Tabular Statistics

TKPROF lists the statistics for a SQL statement returned by the SQL trace facility in rows and columns. Each row corresponds to one of three steps of SQL statement processing. The step for which each row contains statistics is identified by the value of the CALL column:

PARSE	This step translates the SQL statement into an execution plan. This step includes checks for proper security authorization and checks for the existence of tables, columns, and other referenced objects.
EXECUTE	This step is the actual execution of the statement by Oracle. For INSERT, UPDATE, and DELETE statements, this step modifies the data. For SELECT statements, the step identifies the selected rows.
FETCH	This step 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. These values are zero (0) if

TIMED_STATISTICS is not turned on. The sum of *query* and *current* is the total number of buffers accessed.

COUNT	Number of times a statement was parsed, executed, or fetched.
CPU	Total CPU time in seconds for all parse, execute, or fetch calls for the statement.
ELAPSED	Total elapsed time in seconds for all parse, execute, or fetch calls for the statement.
DISK	Total number of data blocks physically read from the datafiles on disk for all parse, execute, or fetch calls.
QUERY	Total number of buffers retrieved in consistent mode for all parse, execute, or fetch calls. Buffers are usually retrieved in consistent mode for queries.
CURRENT	Total number of buffers retrieved in current mode. Buffers are retrieved in current mode for statements such as INSERT, UPDATE, and DELETE.

Rows

Statistics about the processed rows appear in the ROWS column.

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

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

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

Resolution of Statistics

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

Recursive Calls

Sometimes in order to execute a SQL statement issued by a user, Oracle must issue additional statements. Such statements are called *recursive calls* or *recursive SQL statements*. For example, if you insert a row into a table that does not have enough space to hold that row, 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, 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 recursive calls in the output file by setting the SYS statement-line parameter to NO. The statistics for a recursive SQL statement are included in the listing for that statement, not in the listing for the SQL statement that caused the recursive call. So when you are calculating the total resources required to process a SQL statement, you should consider the statistics for that statement as well as those for recursive calls caused by that statement.

Library Cache Misses

TKPROF also lists the number of library cache misses resulting from parse and execute steps for each SQL statement. These statistics appear on separate lines following the tabular statistics. If the statement resulted in no library cache misses, TKPROF does not list the statistic. In "Sample TKPROF Output" on page 14-6, the example, the statement resulted in one library cache miss for the parse step and no misses for the execute step.

Statement Truncation

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

SET ROLE
GRANT
ALTER USER
ALTER ROLE

CREATE USER
CREATE ROLE

User Issuing the SQL Statement

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

Execution Plan

If you specify the EXPLAIN parameter on the TKPROF statement line, 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 13, "Using EXPLAIN PLAN"for more information on interpreting execution plans.

Deciding Which Statements to Tune

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       2
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

10  user SQL statements in session.
 0  internal SQL statements in session.
10  SQL statements in session.
```

If it is acceptable to expend 7.01 CPU seconds to insert, update or delete 2 rows 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 from this summary that 1 unnecessary parse call was made (because there were 11 parse calls, but only 10 user SQL statements) and that array fetch operations were performed. (You know this because more rows were fetched than there were fetches performed.)

Finally, very little physical I/O was performed; this is suspicious and probably means that the same database blocks were being continually revisited.

Having established that the process has used excessive resource, the next step is to discover which SQL statements are the culprits. Normally only a small percentage of the SQL statements in any process need to be tuned—those that use the greatest resource.

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

The SQL parsed as SYS is recursive SQL used to maintain the dictionary cache, and is not normally of great benefit. If the number of internal SQL statements looks high, you might want to check to see what has been going on. (There may be excessive space management overhead.)

Step 5: Store SQL Trace Facility Statistics

This section covers:

- Generating the TKPROF Output SQL Script
- Editing the TKPROF Output SQL Script
- Querying the Output Table

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

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

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

Generating the TKPROF Output SQL Script

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

Editing the TKPROF Output SQL Script

After TKPROF has created the SQL script, you may want to edit the script before running it.

If you have already created an output table for previously collected statistics and you want to add new statistics to this table, remove the CREATE TABLE statement from the script. The script will then insert the new rows into the existing table.

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

Querying the Output Table

The following CREATE TABLE statement creates the TKPROF_TABLE:

```
CREATE TABLE tkprof_table
 (date_of_insert    DATE,
 cursor_num        NUMBER,
 depth             NUMBER,
 user_id           NUMBER,
 parse_cnt         NUMBER,
 parse_cpu         NUMBER,
 parse_elap        NUMBER,
 parse_disk        NUMBER,
 parse_query       NUMBER,
 parse_current     NUMBER,
 parse_miss        NUMBER,
 exe_count         NUMBER,
 exe_cpu           NUMBER,
 exe_elap          NUMBER,
 exe_disk          NUMBER,
 exe_query         NUMBER,
 exe_current       NUMBER,
 exe_miss          NUMBER,
 exe_rows          NUMBER,
 fetch_count       NUMBER,
 fetch_cpu         NUMBER,
 fetch_elap        NUMBER,
 fetch_disk        NUMBER,
 fetch_query       NUMBER,
 fetch_current     NUMBER,
 fetch_rows        NUMBER,
 clock_ticks       NUMBER,
 sql_statement     LONG);
```

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

These columns help you identify a row of statistics:

SQL_STATEMENT	The column value 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	The column value 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 column value indicates the level of recursion at which the SQL statement was issued. For example, a value of 1 indicates that a user issued the statement. A value of 2 indicates Oracle generated the statement as a recursive call to process a statement with a value of 1 (a statement issued by a user). A value of *n* indicates Oracle generated the statement as a recursive call to process a statement with a value of *n*-1.
USER_ID	This column value 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 14-6.

```
SELECT * FROM tkprof_table;
```

Oracle responds with something similar to:

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

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

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

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

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

Avoiding Pitfalls in TKPROF Interpretation

This section describes some fine points of TKPROF interpretation:

- Finding Which Statements Constitute the Bulk of the Load
- The Argument Trap
- The Read Consistency Trap
- The Schema Trap
- The Time Trap
- The Trigger Trap
- The "Correct" Version

See Also: "EXPLAIN PLAN Restrictions" on page 13-28 for information about TKPROF and bind variables.

Finding Which Statements Constitute the Bulk of the Load

Look at the totals and try to identify the statements that constitute the bulk of the load.

Do not attempt to perform many different jobs within a single query. It is more effective to separate out the different queries that should be used when particular optional parameters are present, and when the parameters provided contain wild cards.

If particular parameters are not specified by the report user, the query uses bind variables that have been set to "%". This action has the effect of ignoring any LIKE clauses in the query. It would be more efficient to run a query in which these clauses are not present.

Note: TKPROF cannot determine the TYPE of the bind variables from the text of the SQL statement. It assumes that TYPE is CHARACTER. If this is not the case, you should put appropriate type conversions in the SQL statement.

The Argument Trap

If you are not aware of the values being bound at run time, it is possible to fall into the "argument trap". Especially where the LIKE operator is used, the query may be markedly less efficient for particular values, or types of value, in a bind variable.

This is because the optimizer must make an assumption about the probable selectivity without knowing the value.

The Read Consistency Trap

The next example illustrates the read consistency trap. Without knowing that an uncommitted transaction had made a series of updates to the NAME column it is very difficult to see why so many block visits would be incurred.

Cases like this are not normally repeatable: if the process were run again, it is unlikely that another transaction would interact with it in the same way.

```
select NAME_ID
from CQ_NAMES where NAME = ’FLOOR’;

call     count     cpu     elapsed     disk     query current     rows
----     -----     ---     -------     ----     ----- -------     ----
Parse        1    0.10        0.18        0         0       0        0
Execute      1    0.00        0.00        0         0       0        0
Fetch        1    0.11        0.21        2       101       0        1

Misses in library cache during parse: 1
Parsing user id: 01 (USER1)

Rows     Execution Plan
----     --------- ----
   0     SELECT STATEMENT
   1       TABLE ACCESS (BY ROWID) OF ’CQ_NAMES’
   2         INDEX (RANGE SCAN) OF ’CQ_NAMES_NAME’ (NON_UNIQUE)
```

The Schema Trap

This example shows an extreme (and thus easily detected) example of the schema trap. At first it is difficult to see why such an apparently straightforward indexed query needs to look at so many database blocks, or why it should access any blocks at all in current mode.

```
select NAME_ID
from CQ_NAMES where NAME = ’FLOOR’;

call       count        cpu      elapsed     disk  query current rows
--------  -------   --------    ---------  ------- ------ ------- ----
Parse          1       0.06         0.10        0      0       0    0
Execute        1       0.02         0.02        0      0       0    0
Fetch          1       0.23         0.30       31     51       3    1
```

```
Misses in library cache during parse: 0
Parsing user id: 02  (USER2)

Rows     Execution Plan
-------  ---------------------------------------------------
      0  SELECTSTATEMENT
   2340    TABLE ACCESS (BY ROWID) OF ’CQ_NAMES’
      0      INDEX (RANGE SCAN) OF ’CQ_NAMES_NAME’ (NON-UNIQUE)
```

Two statistics suggest that the query may have been executed via 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.

The Time Trap

Sometimes, as in the following example, you may wonder why a particular query has taken so long.

```
update CQ_NAMES set ATTRIBUTES = lower(ATTRIBUTES)
where ATTRIBUTES = :att

call      count       cpu    elapsed     disk    query current        rows
-------- -------  --------  --------- -------- -------- -------  ----------
Parse          1      0.06       0.24        0        0       0           0
Execute        1      0.62      19.62       22      526      12           7
Fetch          0      0.00       0.00        0        0       0           0

Misses in library cache during parse: 1
Parsing user id: 02  (USER2)

Rows     Execution Plan
-------  ---------------------------------------------------
      0  UPDATE STATEMENT
   2519  TABLE ACCESS (FULL) OF ’CQ_NAMES’
```

Again, the answer is interference from another transaction. In this case another transaction held a shared lock on the table CQ_NAMES for several seconds before and after the update was issued. It takes a fair amount of experience to diagnose that interference effects are occurring. On the one hand, comparative data is essential when the interference is contributing only a short delay (or a small increase in block visits in the previous example). On the other hand, if the interference is contributing only a modest overhead, and the statement is essentially efficient, its statistics may never have to be subjected to analysis.

The Trigger Trap

The resources reported for a statement include those for all of the SQL issued while the statement was being processed. Therefore, they include any resources used within a trigger, along with the resources used by any other recursive SQL (such as that used in space allocation). With the SQL trace facility enabled, TKPROF reports these resources twice. Avoid trying to tune the DML statement if the resource is actually being consumed at a lower level of recursion.

You may need to inspect the raw trace file to see exactly where the resource is being expended. The entries for recursive SQL follow the PARSING IN CURSOR entry for the user's statement. Within the trace file, the order is less easily defined.

The "Correct" Version

For comparison with the output that results from one of the foregoing traps, here is the TKPROF output for the indexed query with the index in place and without any contention effects.

```
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 both elapsed and CPU time, but the query apparently took no time at all to execute and no time at all to perform the fetch. In fact, no parse took place because the query was already available in parsed form within the shared SQL area.

These anomalies arise because the clock tick of 10 msec is too long to reliably record simple and efficient queries.

TKPROF Output Example

This section provides an extensive example of TKPROF output. Portions have been edited out for the sake of brevity.

Header

```
Copyright (c) Oracle Corporation 1979, 1998. All rights reserved.
Trace file: v80_ora_2758.trc
Sort options: default
********************************************************************************
count    = number of times OCI procedure was executed
cpu      = cpu time in seconds executing
elapsed  = elapsed time in seconds executing
disk     = number of physical reads of buffers from disk
query    = number of buffers gotten for consistent read
current  = number of buffers gotten in current mode (usually for update)
rows     = number of rows processed by the fetch or execute call
********************************************************************************
The following statement encountered a error during parse:
select deptno, avg(sal) from emp e group by deptno
       having exists (select deptno from dept
      where dept.deptno = e.deptno
      and dept.budget > avg(e.sal)) order by 1
Error encountered: ORA-00904
********************************************************************************
```

Body

```
alter session set sql_trace = true
call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        0      0.00       0.00          0          0          0           0
Execute      1      0.00       0.10          0          0          0           0
Fetch        0      0.00       0.00          0          0          0           0
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total        1      0.00       0.10          0          0          0           0
Misses in library cache during parse: 0
Misses in library cache during execute: 1
Optimizer goal: CHOOSE
Parsing user id: 02   (USER02)
```

```
********************************************************************************
select emp.ename, dept.dname from emp, dept
  where emp.deptno = dept.deptno

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.11       0.13          2          0          1           0
Execute      1      0.00       0.00          0          0          0           0
Fetch        1      0.00       0.00          2          2          4          14
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total        3      0.11       0.13          4          2          5          14
Misses in library cache during parse: 1
Optimizer goal: CHOOSE
Parsing user id: 02   (USER02)
Rows     Execution Plan
-------  ---------------------------------------------------
      0  SELECT STATEMENT   GOAL: CHOOSE
     14   MERGE JOIN
      4    SORT (JOIN)
      4     TABLE ACCESS (FULL) OF 'DEPT'
     14    SORT (JOIN)
     14     TABLE ACCESS (FULL) OF 'EMP'

********************************************************************************
select a.ename name, b.ename manager from emp a, emp b
  where a.mgr = b.empno(+)

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.01       0.01          0          0          0           0
Execute      1      0.00       0.00          0          0          0           0
Fetch        1      0.01       0.01          1         50          2          14
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total        3      0.02       0.02          1         50          2          14
Misses in library cache during parse: 1
Optimizer goal: CHOOSE
Parsing user id: 01  (USER01)
Rows     Execution Plan
-------  ---------------------------------------------------
      0  SELECT STATEMENT   GOAL: CHOOSE
     13   NESTED LOOPS (OUTER)
     14    TABLE ACCESS (FULL) OF 'EMP'
     13    TABLE ACCESS (BY ROWID) OF 'EMP'

    26     INDEX (RANGE SCAN) OF 'EMP_IND' (NON-UNIQUE)
```

```
********************************************************************************
select  ename,job,sal
from    emp
where  sal =
       (select  max(sal)
        from    emp)

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.00       0.00          0          0          0           0
Execute      1      0.00       0.00          0          0          0           0
Fetch        1      0.00       0.00          0         12          4           1
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total        3      0.00       0.00          0         12          4           1
Misses in library cache during parse: 1
Optimizer goal: CHOOSE
Parsing user id: 01  (USER01)
Rows     Execution Plan
-------  ---------------------------------------------------
      0  SELECT STATEMENT   GOAL: CHOOSE
     14   FILTER
     14    TABLE ACCESS (FULL) OF 'EMP'
     14    SORT (AGGREGATE)
     14     TABLE ACCESS (FULL) OF 'EMP'
********************************************************************************
select  deptno
from    emp
where   job = 'clerk'
group by deptno
having count(*) >= 2

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.00       0.00          0          0          0           0
Execute      1      0.00       0.00          0          0          0           0
Fetch        1      0.00       0.00          0          1          1           0
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total        3      0.00       0.00          0          1          1           0
Misses in library cache during parse: 13
Optimizer goal: CHOOSE
Parsing user id: 01  (USER01)
Rows     Execution Plan
-------  ---------------------------------------------------
      0  SELECT STATEMENT   GOAL: CHOOSE
      0   FILTER
      0    SORT (GROUP BY)
     14     TABLE ACCESS (FULL) OF 'EMP'
********************************************************************************
select  dept.deptno,dname,job,ename
from    dept,emp
where   dept.deptno = emp.deptno(+)
order by dept.deptno

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.00       0.00          0          0          0           0
Execute      1      0.00       0.00          0          0          0           0
Fetch        1      0.00       0.00          0          3          3          10
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total        3      0.00       0.00          0          3          3          10
Misses in library cache during parse: 1
Optimizer goal: CHOOSE
Parsing user id: 01 (USER01)
Rows     Execution Plan
-------  ---------------------------------------------------
      0  SELECT STATEMENT   GOAL: CHOOSE
     14   MERGE JOIN (OUTER)
      4    SORT (JOIN)
      4     TABLE ACCESS (FULL) OF 'DEPT'
     14    SORT (JOIN)
     14     TABLE ACCESS (FULL) OF 'EMP'
********************************************************************************
select grade,job,ename,sal
from   emp,salgrade
where  sal between losal and hisal
order by grade,job

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.04       0.06          2         16          1           0
Execute      1      0.00       0.00          0          0          0           0
Fetch        1      0.01       0.01          1         10         12          10
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total        3      0.05       0.07          3         26         13          10
Misses in library cache during parse: 1
Optimizer goal: CHOOSE
Parsing user id: 02   (USER02)
Rows     Execution Plan
-------  ---------------------------------------------------
      0  SELECT STATEMENT   GOAL: CHOOSE
     14   SORT (ORDER BY)
     14    NESTED LOOPS
      5     TABLE ACCESS (FULL) OF 'SALGRADE'
     70     TABLE ACCESS (FULL) OF 'EMP'
********************************************************************************
select  lpad(' ',level*2)||ename org_chart,level,empno,mgr,job,deptno
from    emp
connect by prior empno = mgr
start   with ename = 'clark'
  or    ename = 'blake'
order   by deptno

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.01       0.01          0          0          0           0
Execute      1      0.00       0.00          0          0          0           0
Fetch        1      0.01       0.01          0          1          2           0
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total        3      0.02       0.02          0          1          2           0
Misses in library cache during parse: 1
Optimizer goal: CHOOSE
Parsing user id: 02   (USER02)
Rows     Execution Plan
-------  ---------------------------------------------------
      0  SELECT STATEMENT   GOAL: CHOOSE
      0   SORT (ORDER BY)
      0    CONNECT BY
     14     TABLE ACCESS (FULL) OF 'EMP'
      0     TABLE ACCESS (BY ROWID) OF 'EMP'
      0     TABLE ACCESS (FULL) OF 'EMP'
********************************************************************************
create table tkoptkp (a number, b number)

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.00       0.00          0          0          0           0
Execute      1      0.01       0.01          1          0          1           0
Fetch        0      0.00       0.00          0          0          0           0
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total        2      0.01       0.01          1          0          1           0
Misses in library cache during parse: 1
Optimizer goal: CHOOSE
Parsing user id: 02   (USER02)
Rows     Execution Plan
-------  ---------------------------------------------------
```

```
      0  CREATE TABLE STATEMENT   GOAL: CHOOSE

********************************************************************************
insert into tkoptkp
values
 (1,1)

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.07       0.09          0          0          0           0
Execute      1      0.01       0.20          2          2          3           1
Fetch        0      0.00       0.00          0          0          0           0
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total        2      0.08       0.29          2          2          3           1
Misses in library cache during parse: 1
Optimizer goal: CHOOSE
Parsing user id: 02   (USER02)
Rows     Execution Plan
-------  ---------------------------------------------------
      0  INSERT STATEMENT   GOAL: CHOOSE
.
********************************************************************************
insert into tkoptkp select * from tkoptkp

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.00       0.00          0          0          0           0
Execute      1      0.02       0.02          0          2          3          11
Fetch        0      0.00       0.00          0          0          0           0
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total        2      0.02       0.02          0          2          3          11
Misses in library cache during parse: 1
Optimizer goal: CHOOSE
Parsing user id: 02   (USER02)
Rows     Execution Plan
-------  ---------------------------------------------------
      0  INSERT STATEMENT   GOAL: CHOOSE
     12   TABLE ACCESS (FULL) OF 'TKOPTKP'
********************************************************************************
select *
from
 tkoptkp where a > 2

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
```

```
Parse        1      0.01       0.01          0          0          0           0
Execute      1      0.00       0.00          0          0          0           0
Fetch        1      0.00       0.00          0          1          2          10
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total        3      0.01       0.01          0          1          2          10
Misses in library cache during parse: 1
Optimizer goal: CHOOSE
Parsing user id: 02   (USER02)
Rows     Execution Plan
-------  ---------------------------------------------------
      0  SELECT STATEMENT   GOAL: CHOOSE
     24   TABLE ACCESS (FULL) OF 'TKOPTKP'
********************************************************************************
```

Summary

```
OVERALL TOTALS FOR ALL NON-RECURSIVE STATEMENTS
call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse       18      0.40       0.53         30        182          3           0
Execute     19      0.05       0.41          3          7         10          16
Fetch       12      0.05       0.06          4        105         66          78
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total       49      0.50       1.00         37        294         79          94
Misses in library cache during parse: 18
Misses in library cache during execute: 1
OVERALL TOTALS FOR ALL RECURSIVE STATEMENTS
call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse       69      0.49       0.60          9         12          8           0
Execute    103      0.13       0.54          0          0          0           0
Fetch      213      0.12       0.27         40        435          0         162
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total      385      0.74       1.41         49        447          8         162
Misses in library cache during parse: 13
   19  user  SQL statements in session.
   69  internal SQL statements in session.
   88  SQL statements in session.
   17  statements EXPLAINed in this session.
********************************************************************************
Trace file: v80_ora_2758.trc
Trace file compatibility: 7.03.02
Sort options: default
       1  session in tracefile.
      19  user  SQL statements in trace file.
      69  internal SQL statements in trace file.
      88  SQL statements in trace file.
      41  unique SQL statements in trace file.
      17  SQL statements EXPLAINed using schema:
           SCOTT.prof$plan_table
             Default table was used.
             Table was created.
             Table was dropped.
    1017  lines in trace file.
```

15

Using Oracle Trace

This chapter describes how to use Oracle Trace to collect Oracle server event data. It covers:

- Introduction to Oracle Trace
- Using Oracle Trace Manager
- Using Oracle Trace Data Viewer
- Manually Collecting Oracle Trace Data

Introduction to Oracle Trace

Oracle Trace is a general-purpose data collection product and is part of the Oracle Enterprise Manager systems management product family. The Oracle server uses Oracle Trace to collect performance and resource utilization data such as SQL Parse, Execute, Fetch statistics, and Wait statistics.

> **See Also:** *Oracle Trace User's Guide* and *Oracle Trace Developer's Guide* in the Oracle Diagnostics Pack documentation set. These books contain a complete list of events and data that you can collect for the Oracle server as well as information on how to implement tracing in your own products and applications.

Using Oracle Trace Data

Among the many advantages of using Oracle Trace is the integration of Oracle Trace with many other applications. You can use Oracle Trace data collected for the Oracle server in the following applications as shown in Figure 15–1:

- Oracle Expert

  You can use information collected with Oracle Trace as an optional source of SQL workload data in Oracle Expert. This SQL data is used when recommending the addition or removal of indexes. Refer to the *Oracle Tuning Pack* documentation for additional information.

- Oracle Trace Data Viewer

  Oracle Trace Data Viewer is a simple viewer for inspecting Oracle Trace collections containing SQL and Wait statistics. You can export Oracle Trace Data to the following products for further analysis:

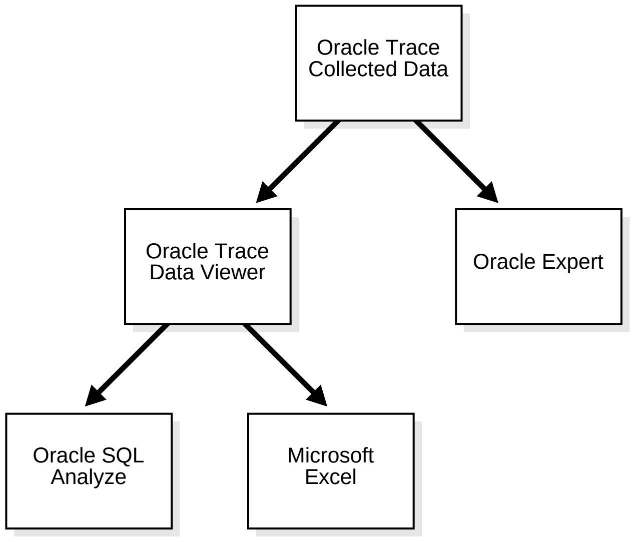
  – SQL Analyze

    You can select one or more rows in Data Viewer and save the SQL statement text to a file that you can import into SQL Analyze. You can then use SQL Analyze to tune these individual statements.

  – Microsoft Excel

    SQL in Data Viewer can be saved to a CSV (Comma Separated Value) file for viewing in Microsoft Excel.

Figure 15–1 Integration of Oracle Trace with Other Applications



Importing Oracle Trace Data into Oracle Expert

You can use Oracle Trace to collect workload data for use in the Oracle Expert application. Oracle Trace collects resource utilization statistics for SQL statements executing against a database in real time. Oracle Trace allows you to collect data about all the SQL statements executing against a database *during* periods of poor performance.

You control the duration of an Oracle Trace collection period. To obtain SQL workload data for a 15-minute period of poor performance, stop collection immediately after the poor performance interval ends.

Importing Data Viewer SQL Into Oracle SQL Analyze

While using Data Viewer, you can select one or more rows in the top portion of the Data View window to save to a file. When you choose SQL (SQL Analyze Format) from File/Save, a file containing query text will be saved. You can then import this *.sql file into Oracle SQL Analyze for tuning of the selected statements.

Oracle SQL Analyze can show you the execution plan for individual queries and let you experiment with various optimizer modes and hints.

Importing Data Viewer Information into Microsoft Excel

While using Data Viewer, you can select one or more rows in the top portion of the Data View window to save to a file. When you choose the CSV file format, Oracle Trace creates a *.csv file that you can load into a Microsoft Excel spreadsheet.

Using Oracle Trace Manager

Oracle Trace provides a graphical Oracle Trace Manager application to create, schedule, and administer Oracle Trace collections for products containing Oracle Trace calls.

The Oracle server has been coded with Oracle Trace API calls to collect both SQL and Wait statistics with a minimum of overhead. Using the Oracle Trace Manager graphical user interface you can:

- Schedule collections
- Filter collections by user
- Filter collections by type of Wait event.
- Format collected data to database tables to preserve historical data
- View SQL and Wait statistics using Oracle Trace Data Viewer

Managing Collections

Use and control of Oracle Trace revolves around the concept of a "collection." A collection is data collected for events that occurred while an instrumented product was running.

With the Oracle Trace Manager, you can schedule and manage collections. When creating a collection, you define the attributes of the collection, such as the collection name, the products and event sets to be included in the collection, and the start and end times. The Oracle Trace Manager includes a Collection Wizard that facilitates the creation and execution of collections.

Once you create a collection you can execute it immediately or schedule it to execute at a specific time or at specified intervals. When a collection executes, it produces a file containing the event data for the products participating in the collection. You can also use a collection as a template for creating other similar collections.

Collecting Event Data

An event is the occurrence of some activity within a product. Oracle Trace collects data for predefined events occurring within a software product created with the Oracle Trace API. That is, the product is embedded with Oracle Trace API calls. An example of an event is a parse or fetch.

There are two types of events:

- Point events

  Point events represent an instantaneous occurrence of something in the instrumented product. An example of a point event is an error occurrence.

- Duration events

  Duration events have a beginning and ending. An example of a duration event is a transaction. Duration events can have other events occur within them; for example, an error can occur *within* a transaction.

The Oracle server is instrumented for 13 events. Three of these events are:

- Database Connection: A point event that records data such as the server login user name.
- SQL Parse: One of the series of SQL processing duration events. This event records a large set of data such as sorts, resource use, and cursor numbers.
- RowSource: Data about the execution plan, such as SQL operation, position, object identification, and number of rows processed by a single row source within an execution plan.

Accessing Collected Data

During a collection, Oracle Trace buffers event data in memory and periodically writes it to a collection binary file. This method ensures low overhead associated with the collection process. You can access event data collected in the binary file by formatting the data to predefined tables which makes the data available for fast, flexible access. These predefined tables are called "Oracle Trace formatter tables."

Oracle Trace Manager provides a mechanism for formatting collection data immediately after a collection or at a later time.

When formatting a collection, you identify the database where Oracle Trace Manager creates the formatted collection as follows:

1. Using Oracle Trace Manager, select a collection to format.

2. Choose the Format command.
3. Specify a target database where the data is to reside.

The collection you select determines which collection definition file and data collection file will be used. The formatted target database determines where the formatted collection data will be stored.

Once the data is formatted, you can access the data using the Data Viewer or by using SQL reporting tools and scripts.

Also, you can access event data by running the Detail report from the Oracle Trace reporting utility. This report provides a basic mechanism for viewing a collection's results. You have limited control over what data is reported and how it is presented.

See Also: The *Oracle Trace Developer's Guide* in the Oracle Diagnostics Pack documentation set for additional information about predefined SQL scripts and the Detail report.

Using Oracle Trace Data Viewer

After using Oracle Trace to collect data, run the Oracle Trace Data Viewer by selecting "View Formatted Data..." from the Oracle Trace Collection menu. Or you can select it directly from the Oracle Diagnostics Pack toolbar. Data Viewer can compute SQL and Wait statistics and resource utilization metrics from the raw data that is collected. Once Data Viewer computes statistics, targeting resource intensive SQL becomes a much simpler task.

Data Viewer computes SQL statistics from data collected by Oracle Trace Manager for all executions of a query during the collection period. Resource utilization during a single execution of a SQL statement may be misleading due to other concurrent activities on the database or node. Combining statistics for all executions may lend a clearer picture about the typical resource utilization occurring when a given query is executed.

Note: You can omit recursive SQL from all data views.

Oracle Trace Predefined Data Views

SQL and Wait statistics are presented in a comprehensive set of Oracle Trace predefined data views. Within Wait statistics, a data view is the definition of a query into the data collected by Oracle Trace. A data view consists of items or statistics to be returned and optionally a sort order and limit of rows to be returned.

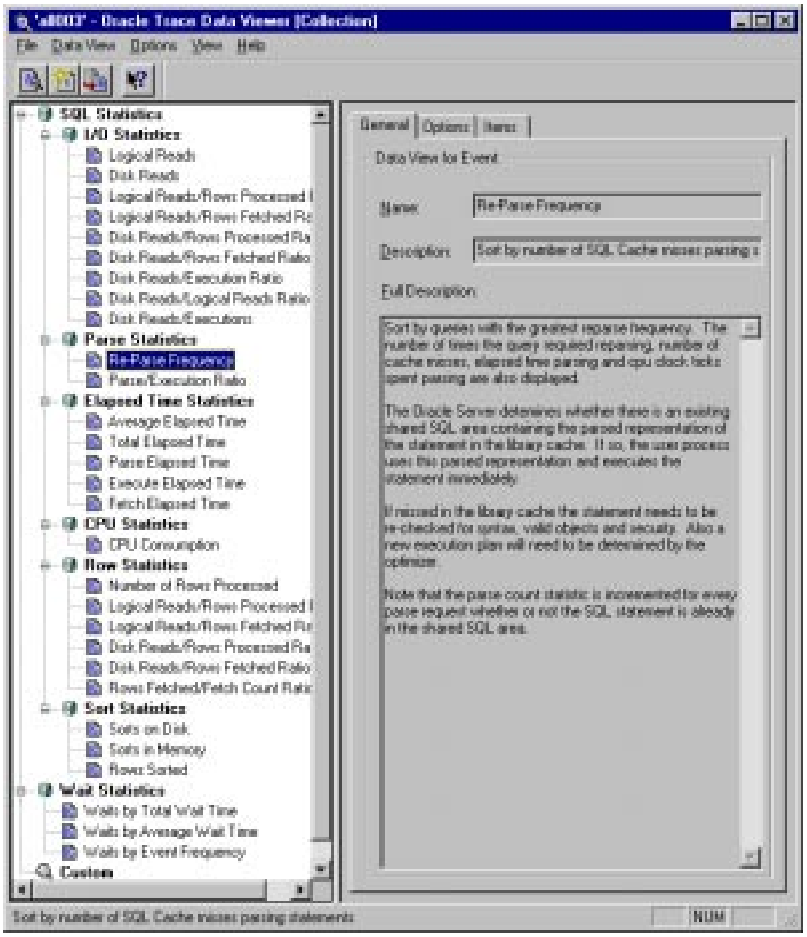
With the data views provided by Data Viewer, you can:

- Examine important statistical data, for example, elapsed times or disk-to-logical-read hit rates.
- Drill down as needed to get additional details about the statement's execution.

In addition to the predefined data views, you can define your own data views using the Oracle Trace Data View Wizard.

Once Data Viewer has computed SQL and Wait statistics, a dialog box showing the available data views appears. SQL Statistic data views are grouped by I/O, Parse, Elapsed Time, CPU, Row, Sort, and Wait statistics as shown in Figure 15–2. A description of the selected data view is shown on the right-hand side of the screen.

Figure 15–2 Oracle Trace Data Viewer - Collection Screen



Data View for Event

Name: Re-Parse Frequency

Description: Sort by number of SQL Cache misses parsing s

Full Description:

Sort by queries with the greatest reparse frequency. The number of times the query required reparsing, number of cache misses, elapsed time parsing and cpu clock ticks spent parsing are also displayed.

The Oracle Server determines whether there is an existing shared SQL area containing the parsed representation of the statement in the library cache. If so, the user process uses this parsed representation and executes the statement immediately.

If missed in the library cache the statement needs to be re-checked for syntax, valid objects and security. Also a new execution plan will need to be determined by the optimizer.

Note that the parse count statistic is incremented for every parse request whether or not the SQL statement is already in the shared SQL area.

Sort by number of SQL Cache misses parsing statements

Table 15–1 explains the predefined data views shown in the previous figure as provided by Oracle Trace.

Table 15–1 Predefined Data Views Provided By Oracle Trace

View Name	Sort By	Data Displayed	Description
Logical Reads	Total number of logical reads performed for each distinct query.	Total number of blocks read during parses, executions and fetches. Logical reads for parses, executions and fetches of the query.	Logical data block reads include data block reads from both memory and disk. Input/output is one of the most expensive operations in a database system. I/O intensive statements can monopolize memory and disk usage causing other database applications to compete for these resources.
Disk Reads	Queries that incur the greatest number of disk reads.	Disk reads for parses, executions, and fetches.	Disk reads also known as physical I/O are database blocks read from disk. The disk read statistic is incremented once per block read regardless of whether the read request was for a multiblock read or a single block read. Most physical reads load data, index, and rollback blocks from the disk into the buffer cache. A physical read count can indicate a high miss rate within the data buffer cache.
Logical Reads/Rows Fetched Ratio	Number of logical reads divided by the number of rows fetched for all executions of the current query.	Total logical I/O. Total number of rows fetched.	The more blocks accessed relative to the number of rows actually returned the more expensive each row is to return. Can be a rough indication of relative expense of a query.
Disk Reads/Rows Fetched Ratio	Number of disk reads divided by the number of rows fetched for all executions of the current query.	Total disk I/O. Total number of rows fetched.	The greater the number of blocks read from disk for each row returned the more expensive each row is to return. Can be a rough indication of relative expense of a query.
Disk Reads/Execution Ratio	Total number of disk reads per distinct query divided by the number of executions of that query.	Total disk I/O. Logical I/O for the query as well as the number of executions of the query.	Indicates which statements incur the greatest number of disk reads per execution.
Disk Reads/Logical Reads Ratio	Greatest miss rate ratio of disk to logical reads.	Individual logical reads. Disk reads for the query as well as the miss rate.	The miss rate indicates the percentage of times the Oracle server needed to retrieve a database block on disk as opposed to locating it in the data buffer cache in memory. The miss rate for the data block buffer cache is derived by dividing the physical reads by the number of accesses made to the block buffer to retrieve data in consistent mode plus the number of blocks accessed through single block gets. Memory access is much faster than disk access, the greater the hit ratio, the better the performance.

Table 15–1 Predefined Data Views Provided By Oracle Trace

View Name	Sort By	Data Displayed	Description
Re-Parse Frequency	Queries with the greatest reparse frequency.	Number of cache misses. Total number of parses. Total elapsed time parsing. Total CPU clock ticks spent parsing.	The Oracle server determines whether there is an existing shared SQL area containing the parsed representation of the statement in the library cache. If so, the user process uses this parsed representation and executes the statement immediately. If missed in the library cache, re-check the statement for syntax, valid objects, and security. Also, the optimizer must determine a new execution plan. The parse count statistic is incremented for every parse request, regardless of whether the SQL statement is already in the shared SQL area.
Parse/Execution Ratio	Number of parses divided by the number executions per statement.	Individual number of parses. Number of executions.	The count of parses to executions should be as close to one as possible. If there are a high number of parses per execution then the statement has been needlessly reparsed. This could indicate the lack of use of bind variables in SQL statements or poor cursor reuse. Reparsing a query means that the SQL statement has to be re-checked for syntax, valid objects and security. Also a new execution plan will need to be determined by the optimizer.
Average Elapsed Time	Greatest average time spent parsing, executing and fetching on behalf of the query.	Individual averages for parse, execution and fetch.	The average elapsed time for all parses, executions and fetches-per-execution are computed, then summed for each distinct SQL statement in the collection.
Total Elapsed Time	Greatest total elapsed time spent parsing, executing and fetching on behalf of the query.	Individual elapsed times for parses, executions and fetches.	The total elapsed time for all parses, executions and fetches are computed, then summed for each distinct SQL statement in the collection.
Parse Elapsed Time	Total elapsed time for all parses associated with a distinct SQL statement.	SQL cache misses. Elapsed times for execution and fetching. Total elapsed time.	During parsing the Oracle server determines whether there is an existing shared SQL area containing the parsed representation of the statement in the library cache. If so, the user process uses this parsed representation and executes the statement immediately. If missed in the library cache, the statement needs to be rechecked for syntax, valid objects and security. Also a new execution plan will need to be determined by the optimizer.
Execute Elapsed Time	Greatest total elapsed time for all executions associated with a distinct SQL statement.	Total elapsed time. Individual elapsed times for parsing and fetching.	The total elapsed time of all execute events for all occurrences of the query within an Oracle Trace collection.

Table 15–1 Predefined Data Views Provided By Oracle Trace

View Name	Sort By	Data Displayed	Description
Fetch Elapsed Time	Greatest total elapsed time for all fetches associated with a distinct SQL statement.	Number of rows fetched. Number of fetches. Number of executions. Total elapsed time. Individual elapsed times for parsing and executing.	The total elapsed time spent fetching data on behalf of all occurrences of the current query within the Oracle Trace collection.
CPU Statistics	Total CPU clock ticks spent parsing, executing and fetching on behalf of the SQL statement.	CPU clock ticks for parses, executions and fetches. Number of SQL cache misses and sorts in memory.	When SQL statements and other types of calls are made to an Oracle server, a certain amount of CPU time is necessary to process the call. Average calls require a small amount of CPU time. However, a SQL statement involving a large amount of data, a runaway query, in memory sorts or excessive reparsing can potentially consume a large amount of CPU time. CPU time displayed is in terms of the number of CPU clock ticks on the operating system housing the database.
Number of Rows Returned	Greatest total number of rows returned during execution and fetch for the SQL statement.	Number of rows returned during the fetch operation as well as the execution rows.	Targets queries that manipulate the greatest number of rows during fetching and execution. May mean that high gains can be made by tuning row intensive queries.
Rows Fetched/Fetch Count Ratio	Number of rows fetched divided by the number of fetches.	Individual number of rows fetched. Number of fetches.	This ratio shows how many rows were fetched at a time. It may indicate the level to which array fetch capabilities have been utilized. A ratio close to one may indicate an opportunity to optimize code by using array fetches.
Sorts on Disk	Queries that did the greatest number of sorts on disk.	Sort statistics for SQL statements. Number of in memory sorts. Total number of rows sorted.	Sorts on disk are sorts that could not be performed in memory, therefore they are more expensive because memory access is faster than disk access.
Sorts in Memory	Queries that did the greatest number of sorts in memory.	Sort statistics for SQL statements. Number of disk sorts. Total number of rows sorted.	Sorts in memory are sorts that could be performed completely within the sort buffer in memory without using the temporary tablespace segments.
Rows Sorted	Queries that sorted the greatest number of rows.	Number of in memory sorts. Number of sorts on disk.	Returns sort statistics for SQL statements ordered by queries that sorted the greatest number of rows.

Table 15–1 Predefined Data Views Provided By Oracle Trace

View Name	Sort By	Data Displayed	Description
Waits by Total Wait Time	Highest total wait time per distinct type of wait.	Average wait time, total wait time and number of waits per wait type.	Waits are sorted by wait description or type that had the greatest cumulative wait time for all occurrences of the wait type within the collection.
Waits by Average Wait Time	Highest average wait time per wait type.	Average wait time, total wait time and number of waits per wait type.	Waits are sorted by wait description or type that had the greatest average wait time for all occurrences of the wait type within the collection.
Waits by Event Frequency	Frequency of waits per wait type.	Number of waits per wait type, average wait time, and total wait time.	Waits are sorted by wait events or wait descriptions that appear most frequently within the collection.

Viewing Oracle Trace Data

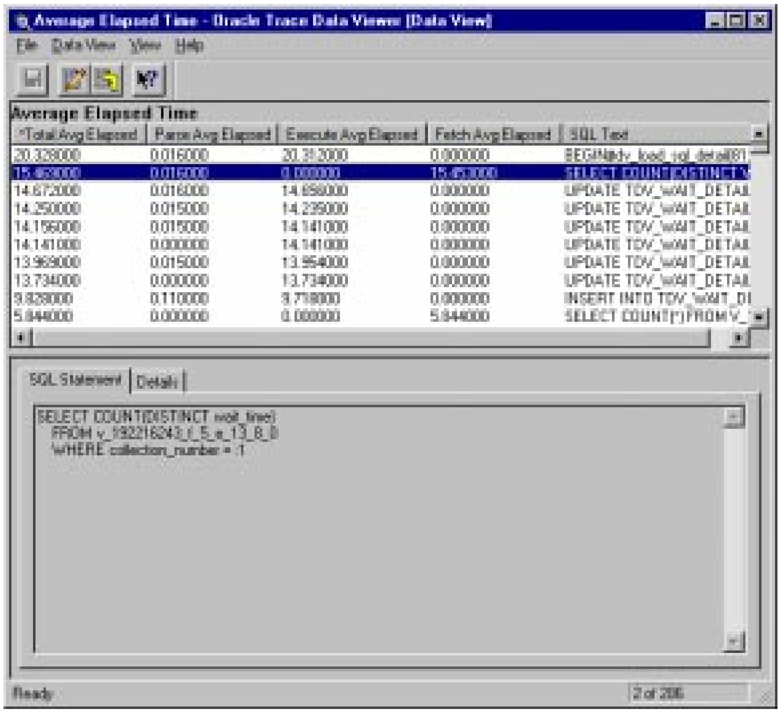
Double clicking on SQL or Wait event data views provided by Data Viewer causes Oracle trace to query the collection data and display data sorted by criteria described in the data view's description.

For example, double clicking the "Disk Reads/Log Reads Ratio" view returns data sorted by queries with the highest data buffer cache miss rate. This also displays the individual disk and logical read values.

Double clicking the "Average Elapsed Time" data view returns data sorted by queries that took the greatest average elapsed time to parse, execute, and fetch. It also displays the average elapsed times for parsing, execution, and fetching.

Figure 15–3 shows data in the "Average Elapsed Time" data view. Query text and statistics appear in the top portion of the window. Clicking any column headers causes the Data Viewer to sort rows by the statistic in that column.

Figure 15–3 Oracle Trace Data Viewer - Data View Screen



*Total Avg Elapsed	Parse Avg Elapsed	Execute Avg Elapsed	Fetch Avg Elapsed	SQL Text
20.328000	0.016000	20.312000	0.000000	BEGIN#tdv_load_sql_detail81
15.469000	0.016000	0.000000	15.453000	SELECT COUNT(DISTINCT w
14.672000	0.016000	14.656000	0.000000	UPDATE TDV_WAIT_DETAIL
14.250000	0.015000	14.235000	0.000000	UPDATE TDV_WAIT_DETAIL
14.156000	0.015000	14.141000	0.000000	UPDATE TDV_WAIT_DETAIL
14.141000	0.000000	14.141000	0.000000	UPDATE TDV_WAIT_DETAIL
13.969000	0.015000	13.954000	0.000000	UPDATE TDV_WAIT_DETAIL
13.734000	0.000000	13.734000	0.000000	UPDATE TDV_WAIT_DETAIL
9.828000	0.110000	9.718000	0.000000	INSERT INTO TDV_WAIT_DI
5.844000	0.000000	0.000000	5.844000	SELECT COUNT(*) FROM V_

```
SELECT COUNT(DISTINCT wait_time)
  FROM v_192216243_f_5_e_13_8_0
  WHERE collection_number = 1
```

The currently selected data view's SQL text is shown in the lower portion of the window in the SQL Statement property sheet. Full statistical details about the currently selected data view also appear in the Details property sheet.

When examining a data view like that shown in Figure 15–3, you can print the following:

- Data view statistics, located in the top portion of the screen, or
- Current SQL statement text in formatted output plus details on all statistical data collected for the currently selected query, located in the Details property page.

Window focus at the time of printing determines which portion of the screen is printed. For example, if focus is on the top portion of the screen, the tabular form of all statistics and SQL for this data view is printed.

SQL Statement Property Page

The SQL Statement property page displays the currently selected query in a formatted output.

Details Property Page

The Details property page displays a detailed report on statistics for all executions of a given query within an Oracle Trace collection. Text for the currently selected SQL statement is posted at the end of the property page.

Example of Details Property Page

Statistics for all parses, executions, and fetches of the SQL statement.

The number of misses in library cache during Parse: 1.000000

Elapsed time statistics for the SQL statement:

```
Average Elapsed Time:     0.843000
Total Elapsed Time:       0.843000

Total Elapsed Parse:      0.000000
Total Elapsed Execute:    0.843000
Total Elapsed Fetch:      0.000000

Average Elapsed Parse:    0.000000
Average Elapsed Execute:  0.843000
Average Elapsed Fetch:    0.000000
```

Number of times parse, execute and fetch were called:

```
Number of Parses:      1
Number of Executions:  1
Number of Fetches:     0
```

Logical I/O statistics for parse, execute and fetch calls:

```
Logical I/O for Parses:             1
Logical I/O for Executions:       247
Logical I/O for Fetches:            0
Logical I/O Total:                  0
```

Disk I/O statistics for parse, execute and fetch calls:

```
Disk I/O for Parses:              0
Disk I/O for Executions:         28
Disk I/O for Fetches:             0
Disk I/O Total:                   0
```

CPU statistics for parse, execute and fetch calls:

```
CPU for Parses:              0
CPU for Executions:      62500
CPU for Fetches:             0
CPU Total:               62500
```

Row statistics for execute and fetch calls:

```
Rows processed during Executions:        104
Rows processed during Fetches:             0
Rows Total:                              104
```

Sort statistics for execute and fetch calls:

```
Sorts on disk:             0
Sorts in memory:           2
Sort rows:               667
```

Hit Rate - Logical I/O divided by Disk I/O: 0.112903

Logical I/O performed divided by rows actually processed: 2.384615

Disk I/O performed divided by number of executions: 28.000000

The number of parses divided by number of executions: 1.000000

The number of rows fetched divided by the number of fetches: 0.000000

```
INSERT INTO tdv_sql_detail
 (collection_number, sql_text_hash,
  "LIB_CACHE_ADDR")
 SELECT DISTINCT collection_number,
                 sql_text_hash,
                 "LIB_CACHE_ADDR"
     FROM v_192216243_f_5_e_7_8_0
     WHERE collection_number = :b1;
```

Getting More Information on a Selected Query

There are two convenient ways to obtain additional data for the currently selected SQL statement:

- To modify a data view to add or remove statistics or items, select Modify from the Data View menu. You may add or remove statistics in the Items property sheet. These statistics appear as new columns in the data view. The selected query in Figure 15–3 is:

  ```
  SELECT COUNT(DISTINCT WAIT_TIME)
  FROM v_192216243_f_5_e_13_8_0
  WHERE collection_number = :1;
  ```

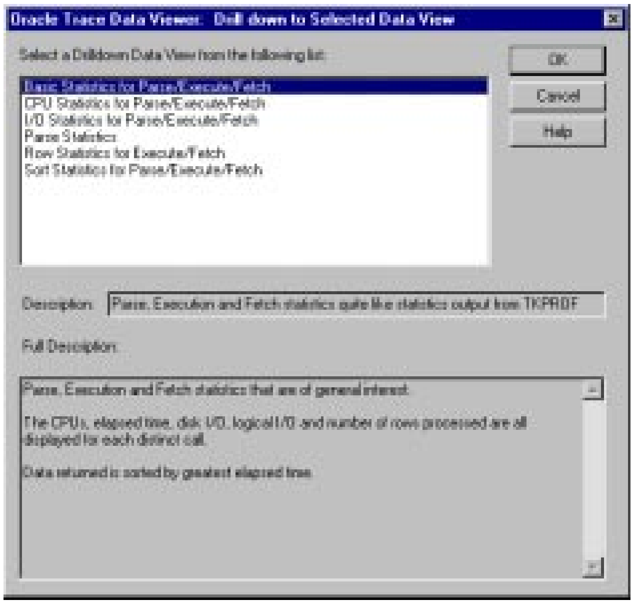
  This query counts distinct values in the WAIT_TIME column of the v_192216243_f_5_e_13_8_0 table. By modifying the existing data view, you can add other statistics that may be of interest such as "Sort Rows", which is the number of rows sorted, or "Execute CPU", which is the number of CPU clock ticks during execution.

  This query counts distinct values in the WAIT_TIME column of the WAITS table. By modifying the existing data view you can add other statistics that may be of interest such as "Execute Rows", which is the number of rows processed during execution, or "Execute CPU", which is the number of CPU clock ticks during execution.

  You can also remove existing columns, change the sort order, or change the default number of rows to view. You can save a modified view to a new, user-defined data view. Oracle stores user-defined data views in the Custom data view container following the Data Viewer supplied list of SQL and Wait data views.

- Drill to statistics on all parses, executions and fetches of the selected query by clicking the Drill icon in the toolbar. The Drill down Data View dialog is displayed as shown in Figure 15–4.

Figure 15–4 Oracle Trace Data Viewer - Drill Down Data View Screen


Oracle Trace Data Viewer: Drill down to Selected Data View

Select a Drilldown Data View from the following list:

Basic Statistics for Parse/Execute/Fetch
CPU Statistics for Parse/Execute/Fetch
I/O Statistics for Parse/Execute/Fetch
Parse Statistics
Row Statistics for Execute/Fetch
Sort Statistics for Parse/Execute/Fetch

Description: Parse, Execution and Fetch statistics quite like statistics output from TKPROF

Full Description:

Parse, Execution and Fetch statistics that are of general interest.

The CPUs, elapsed time, disk I/O, logical I/O and number of rows processed are all displayed for each distinct call.

Data returned is sorted by greatest elapsed time.

Drill-down data views show individual statistics for all parses, executions, and fetches.

In Figure 15–4 the "Basic Statistics for Parse / Execute / Fetch" drill-down data view is selected. It displays statistics similar to those from TKPROF.

Note: For more information on TKPROF, please refer to Chapter 12, "Overview of Diagnostic Tools".

Table 15–2 Drill-down Data Views

Drill-down Name	Sort By	Data Displayed	Description
Basic Statistics for Parse/Execute/Fetch	Greatest elapsed time.	For each distinct call:. CPUs. Elapsed time. Disk I/O. Logical I/O. Number of rows processed.	Parse, Execution, and Fetch statistics which are similar to statistics from TKPROF.
CPU Statistics for Parse/Execute/Fetch	Greatest number of CPUs.	CPU total. Pagefaults.	CPU and pagefault statistics for Parses, Executions, and Fetches of the current query. CPU total is the number of clock ticks in both user and system mode. The clock tick granularity is specific to the operating system on which the database resides.
I/O Statistics for Parse/Execute/Fetch	Greatest number of disk I/Os.	Logical and Disk I/O statistics. Pagefault I/O (number of hard pagefaults). Input I/O (number of times the file system performed input). Output I/O (number of times the file system performed output).	I/O statistics for parses, executions, and fetches.
Parse Statistics	Greatest elapsed time.	Current user identifier. Schema identifiers.	Parse statistics, for example, whether the current statement was missed in library cache, Oracle optimizer mode, current user identifier, and schema identifier.

Table 15–2 Drill-down Data Views

Drill-down Name	Sort By	Data Displayed	Description
Row Statistics for Execute/Fetch	Greatest number of rows returned.	Number of rows returned. Number of rows sorted. Number of rows returned during a full table scan.	Execution and fetch row statistics.
Sort Statistics for Parse/Execute/Fetch	Greatest elapsed time.	Sorts on disk. Sorts in memory. Number of rows sorted. Number of rows returned from a full table scan.	Parse, execution, and fetch sort statistics.
Wait Parameters	Wait_time.	Description. Wait_time. P1. P2. P3.	Investigating waits may help identify sources of contention. P1, P2, and P3 parameters are values that provide more information about specific wait events. The parameters are foreign keys to views that are wait event dependent. For example, for latch waits, P2 is the latch number that is a foreign key to V$LATCH. The meaning of each parameter is specific to each wait type.

Manually Collecting Oracle Trace Data

Though the Oracle Trace Manager is the primary interface to Oracle Trace, you can optionally force a manual collection of Oracle Trace data. You can do this by using a command-line interface, editing initialization parameters, or by executing stored procedures.

Using the Oracle Trace Command-Line Interface

Another option for controlling Oracle Trace server collections is the Oracle Trace CLI (Command-line Interface). The CLI collects event data for all server sessions attached to the database at collection start time. Sessions that attach after the collection is started are excluded from the collection. The CLI is invoked by the OTRCCOL command for the following functions:

- OTRCCOL START job_id input_parameter_file
- OTRCCOL STOP job_id input_parameter_file
- OTRCCOL FORMAT input_parameter_file

- OTRCCOL DCF col_name cdf_file
- OTRCCOL DFD col_name username password service

The parameter JOB_ID can be any numeric value but must be unique and you must remember this value to stop the collection. The input parameter file contains specific parameter values required for each function as shown in the following examples. COL_NAME (collection name) and CDF_FILE (collection definition file) are initially defined in the START function input parameter file.

The OTRCCOL START command invokes a collection based upon parameter values contained in the input parameter file. For example:

```
OTRCCOL  START  1234  my_start_input_file
```

Where file MY_START_INPUT_FILE contains the following input parameters:

col_name	my_collection
dat_file	<usually same as collection name>.dat
cdf_file	<usually same as collection name>.cdf
fdf_file	<server event set>.fdf
regid	1 192216243 0 0 5 <database SID>

The server event sets that can be used as values for the fdf_file are ORACLE, ORACLEC, ORACLED, ORACLEE, and ORACLESM.

See Also: "Using Initialization Parameters to Control Oracle Trace" on page 15-21 for more information on the server event sets.

The OTRCCOL STOP command halts a running collection as follows:

```
OTRCCOL STOP 1234 my_stop_input_file
```

Where my_stop_input_file contains the collection name and cdf_file name.

The OTRCCOL FORMAT command formats the binary collection file to Oracle tables. An example of the FORMAT command is:

```
otrccol format my_format_input_file
```

Where my_format_input_file contains the following input parameters:

username	<database username>
password	<database password>

service	<database service name>
cdf_file	<usually same as collection name>.cdf
full_format	<0/1>

A full_format value of 1 produces a full format; a value of 0 produces a partial format.

> **See Also:** "Formatting Oracle Trace Data to Oracle Tables" on page 15-25 for information on formatting part or all of an Oracle Trace collection and for other important information on creating the Oracle Trace formatting tables prior to running the format command.

The OTRCCOL DCF command deletes collection files for a specific collection. The OTRCCOL DFD command deletes formatted data from the Oracle Trace formatter tables for a specific collection.

Using Initialization Parameters to Control Oracle Trace

Six parameters are set up by default to control Oracle Trace. By logging into the administrator account in your database and executing the SHOW PARAMETERS TRACE command, you will see the following parameters as shown in Table 15–3:

Table 15–3 Oracle Trace Initialization Parameters

Name	Type	Value
ORACLE_TRACE_COLLECTION_NAME	string	[null]
ORACLE_TRACE_COLLECTION_PATH	string	$ORACLE_HOME/otrace/admin/cdf
ORACLE_TRACE_COLLECTION_SIZE	integer	5242880
ORACLE_TRACE_ENABLE	boolean	FALSE
ORACLE_TRACE_FACILITY_NAME	string	oracled
ORACLE_TRACE_FACILITY_PATH	string	$ORACLE_HOME/otrace/admin/cdf

You can modify the Oracle Trace initialization parameters and use them by adding them to your initialization file.

Note: This chapter references file path names on UNIX-based systems. For the exact path on other operating systems, please see your Oracle platform-specific documentation.

See Also: A complete discussion of these parameters is provided in *Oracle8i Reference*.

Enabling Oracle Trace Collections

The ORACLE_TRACE_ENABLE parameter is set to FALSE by default. A value of FALSE disables any use of Oracle Trace for that Oracle server.

To enable Oracle Trace collections for the server, set the parameter to TRUE. Having the parameter set to TRUE does not start an Oracle Trace collection, but instead allows Oracle Trace to be used for that server. You can then start Oracle Trace in one of the following ways:

- Using the Oracle Trace Manager application supplied with the Oracle Diagnostics Pack.
- Setting the ORACLE_TRACE_COLLECTION_NAME parameter.

When ORACLE_TRACE_ENABLE is set to TRUE, you can start and stop an Oracle Trace server collection by either using the Oracle Trace Manager application that is supplied with the Oracle Diagnostics Pack, or you can enter a collection name in the ORACLE_TRACE_COLLECTION_NAME parameter. The default value for this parameter is NULL. A collection name can be up to 16 characters in length. You must then shut down your database and start it up again to activate the parameters. If a collection name is specified, when you start the server, you automatically start an Oracle Trace collection for all database sessions, which is similar in functionality to SQL Trace.

To stop the collection that was started using the ORACLE_TRACE_COLLECTION_NAME parameter, shut down the server instance and reset the ORACLE_TRACE_COLLECTION_NAME to NULL. The collection name specified in this value is also used in two collection output file names: the collection definition file (*collection_name*.cdf) and the binary data file (*collection_name*.dat).

Determining the Event Set that Oracle Trace Collects

The ORACLE_TRACE_FACILITY_NAME initialization parameter specifies the event set that Oracle Trace collects. The name of the DEFAULT event set is

ORACLED. The ALL event set is ORACLE, the EXPERT event set is ORACLEE, the SUMMARY event set is ORACLESM, and the CACHEIO event set is ORACLEC.

If once restarted, the database does not begin collecting data, check the following:

- The event set file, identified by ORACLE_TRACE_FACILITY_NAME, with .fdf appended to it, should be in the directory specified by the ORACLE_TRACE_FACILITY_PATH initialization parameter. The exact directory that this parameter specifies is platform-specific.
- The following files should exist in your Oracle Trace administrative directory: REGID.DAT, PROCESS.DAT, and COLLECT.DAT. If they do not, you must run the OTRCCREF executable to create them.
- The Oracle Trace parameters should be set to the values that you changed in the initialization file. Use Instance Manager to identify Oracle Trace parameter settings.
- Look for an EPC_ERROR.LOG file to see more information about why a collection failed. Oracle Trace creates the EPC_ERROR.LOG file in the current default directory of the Oracle Intelligent Agent when it runs the Oracle Trace Collection Services OTRCCOL image. Depending on whether you are running Oracle Trace from the Oracle Trace Manager or from the command-line interface, you can find the EPC_ERROR.LOG file in one of the following locations:
  - $ORACLE_HOME or $ORACLE_HOME/network/agent (on UNIX)
  - %ORACLE_HOME%\network\agent or %ORACLE_HOME%\net80\agent (on NT)
  - $ORACLE_HOME\rdbmsnn on NT ($ORACLE_HOME\rdbms on UNIX)
  - in current working directory, if you are using the command-line interface
  - To find the EPC_ERROR.LOG file on UNIX, change directories to the $ORACLE_HOME directory and execute the command:

    ```
    find . -name EPC_ERROR.LOG -print .
    ```

    Note: On UNIX, the EPC_ERROR.LOG file name is case sensitive and is in uppercase.

  - Look for *.trc files in the directory specified by the USER_DUMP_DEST initialization parameter. Searching for "epc" in the *.trc files may give errors.

These errors and their descriptions are located in the $ORACLE_HOME/otrace/include/epc.h file.

Using Stored Procedures to Control Oracle Trace

Using the Oracle Trace stored procedures you can invoke an Oracle Trace collection for your own session or for another session. To collect Oracle Trace data for your own database session, execute the following stored procedure package syntax:

```
DBMS_ORACLE_TRACE_USER.SET_ORACLE_TRACE(TRUE/FALSE,
collection_name, serverevent_set)
```

where:

True/false	Boolean: TRUE to activate, FALSE to deactivate.
Collection_name	VARCHAR2: collection name (no file extension, eight character maximum).
Server_event_set	VARCHAR2: server event set (oracled, oracle, or oraclee).

Example:

```
EXECUTE DBMS_ORACLE_TRACE_USER.SET_ORACLE_TRACE (TRUE,'MYCOLL','oracle');
```

To collect Oracle Trace data for a database session other than your own, execute the following stored procedure package syntax:

```
DBMS_ORACLE_TRACE_AGENT.SET_ORACLE_TRACE_IN_SESSION
(sid, serial#, true/false, collection_name, server_event_set)
```

where:

sid	Number: session instance from V$SESSION.SID.
serial#	Number: session serial number from V$SESSION.SERIAL#.

Example:

```
EXECUTE DBMS_ORACLE_TRACE_AGENT.SET_ORACLE_TRACE_IN_SESSION
(8,12,TRUE,'NEWCOLL','oracled');
```

If the collection does not occur, check the following:

- Be sure the server event set file identified by SERVER_EVENT_SET exists. If there is no full file specification on this field, then the file should be located in the directory identified by ORACLE_TRACE_FACILITY_PATH in the initialization file.

- The following files should exist in your Oracle Trace admin directory: REGID.DAT, PROCESS.DAT, and COLLECT.DAT. If they do not, you must run the OTRCCREF executable to create them.
- The stored procedure packages should exist in the database. If the packages do not exist, run the OTRCSVR.SQL file (in your Oracle Trace admin directory) to create the packages.
- The user has the EXECUTE privilege on the stored procedure.

Oracle Trace Collection Results

Running an Oracle Trace collection produces the following collection files:

- collection_name.CDF is the Oracle Trace collection definition file for your collection
- collection_name.DAT files are the Oracle Trace output files containing the trace data in binary format

You can access the Oracle Trace data in the collection files in the following ways:

- You can create Oracle Trace reports from the binary file.
- The data can be formatted to Oracle tables for Data Viewer, SQL access, and reporting.

Formatting Oracle Trace Data to Oracle Tables

You can format Oracle Trace server collection Oracle tables for more flexible access SQL reporting tools. Oracle Trace produces a separate table for each event collected. For example, a parse event table is created to store data for all parse events occurring during a server collection. Before you can format data, you must first set up the Oracle Trace formatter tables by executing the OTRCFMTC.SQL script on the server host machine.

Note: Oracle server releases 7.3.4, 8.0.4, and later, automatically create the formatter tables.

Use the following syntax to format an Oracle Trace collection:

```
OTRCFMT [optional parameters] collection_name.cdf [user/password@database]
```

If you omit user/password@database, Oracle prompts you for this information.

Oracle Trace allows data to be formatted while a collection is occurring. By default, Oracle Trace formats only the portion of the collection that has not been formatted previously. If you want to reformat the entire collection file, use the optional parameter -f.

Oracle Trace provides several SQL scripts that you can use to access the server event tables. For more information on server event tables and scripts for accessing event data and improving event table performance, refer to the *Oracle Trace User's Guide*.

Oracle Trace Statistics Reporting Utility

The Oracle Trace statistics reporting utility displays statistics for all items associated with each occurrence of a server event. These reports can be quite large. You can control the report output by using command parameters. Use the following command and optional parameters to produce a report:

```
OTRCREP  [optional parameters]  collection_name.CDF
```

First, you may want to run a report called "PROCESS.txt". You can produce this report to provide a listing of specific process identifiers for which you want to run another report.

You can manipulate the output of the Oracle Trace reporting utility by using the following optional report parameters:

output_path	Specifies a full output path for the report files. If not specified, the files will be placed in the current directory.
-p	Organizes event data by process. If you specify a process ID (pid), you will have one file with all the events generated by that process in chronological order. If you omit the process ID, you will have one file for each process that participated in the collection. The output files are named *collection*_P*pid*.txt.
-P	Produces a report called *collection*_PROCESS.txt that lists all processes that participated in the collection. It does not include event data. You could produce this report first to determine the specific processes for which you might want to produce more detailed reports.

-w#	Sets report width, such as -w132. The default is 80 characters.
-l#	Sets the number of report lines per page. The default is 63 lines per page.
-h	Suppresses all event and item report headers, producing a shorter report.
-s	Used with Net8 data only. This option creates a file similar to the SQLNet Tracing file.
-a	Creates a report containing all the events for all products, in the order they occur in the data collection (.dat) file.

Part IV
Optimizing Instance Performance

Part IV describes how to tune various elements of your database system to optimize performance of an Oracle instance. The chapters are:

16 Dynamic Performance Views

This chapter explains how you can use V$ views for:

- Instance-Level Views for Tuning
- Session-Level or Transient Views for Tuning
- Current Statistic Values and Rates of Change

> **See Also:** For complete information on all dynamic performance tables, please see the *Oracle8i Reference*.

Dynamic performance views, or "V$" views, are useful for identifying instance-level performance problems. All V$ views are listed in the V$FIXED_TABLE view.

V$ view content is provided by underlying X$ tables. The X$ tables are internal data structures that can be modified by SQL statements. These tables are therefore only available when an instance is in a NOMOUNT or MOUNT state.

This chapter describes the most useful V$ views for performance tuning. V$ views are also useful for ad hoc investigation, for example, when users report sudden response time deterioration.

Although the V$ views belong to user SYS, users other than SYS have read-only access to V$ views. Oracle populates the V$ views and X$ tables at instance startup. Their contents are flushed when you shut down the instance.

The X$ tables and their associated V$ views are dynamic, so their contents are constantly changing. X$ tables retain timing information providing you have set the init.ora parameter TIME_STATISTICS to TRUE, or if you execute the SQL command:

```
ALTER SYSTEM SET TIME_STATISTICS=true;
```

Instance-Level Views for Tuning

These views concern the instance as a whole and record statistics either since startup of the instance or (in the case of the SGA statistics) the current values, which will remain constant until altered by some need to reallocate SGA space. Cumulative statistics are from startup.

Table 16–1 Instance Level Views Important for Tuning

View	Notes
V$FIXED_TABLE	Lists the fixed objects present in the release.
V$INSTANCE	Shows the state of the current instance.
V$LATCH	Lists statistics for nonparent latches and summary statistics for parent latches.
V$LIBRARYCACHE	Contains statistics about library cache performance and activity.
V$ROLLSTAT	Lists the names of all online rollback segments.
V$ROWCACHE	Shows statistics for data dictionary activity.
V$SGA	Contains summary information on the system global area.
V$SGASTAT	Contains detailed information on the system global area.
V$SORT_USAGE	Shows the size of the temporary segments and the session creating them. This information can help you identify which processes are doing disk sorts.
V$SQLAREA	Lists statistics on shared SQL area; contains one row per SQL string. Provides statistics on SQL statements that are in memory, parsed, and ready for execution. Text limited to 1000 characters; full text is available in 64 byte chunks from V$SQLTEXT.
V$SQLTEXT	Contains the text of SQL statements belonging to shared SQL cursors in the SGA.
V$SYSSTAT	Contains basic instance statistics.
V$SYSTEM_EVENT	Contains information on total waits for an event.
V$WAITSTAT	Lists block contention statistics. Updated only when timed statistics are enabled.

The single most important fixed view is V$SYSSTAT, which contains the statistic name in addition to the value. The values from this table form the basic input to the instance tuning process.

Session-Level or Transient Views for Tuning

These views either operate at the session level or primarily concern transient values. Session data is cumulative from connect time.

Table 16–2 Session Level Views Important for Tuning

View	Notes
V$LOCK	Lists the locks currently held by the Oracle8 Server and outstanding requests for a lock or latch.
V$MYSTAT	Shows statistics from your current session.
V$PROCESS	Contains information about the currently active processes.
V$SESSION	Lists session information for each current session. Links SID to other session attributes. Contains row lock information.
V$SESSION_EVENT	Lists information on waits for an event by a session.
V$SESSION_WAIT	Lists the resources or events for which active sessions are waiting, where WAIT_TIME = 0 for current events.
V$SESSTAT	Lists user session statistics. Requires join to V$STATNAME, V$SESSION.

The structure of V$SESSION_WAIT makes it easy to check in real time whether any sessions are waiting, and if so, why. For example:

```
SELECT sid,
  EVENT
  FROM V$SESSION_EVENT
  WHERE WAIT_TIME = 0;
```

You can then investigate to see whether such waits occur frequently and whether they can be correlated with other events, such as the use of particular modules.

Current Statistic Values and Rates of Change

This section describes procedures for:

- Finding the Current Value of a Statistic
- Finding the Rate of Change of a Statistic

Finding the Current Value of a Statistic

Key ratios are expressed in terms of instance statistics. For example, the consistent change ratio is consistent changes divided by consistent gets. The simplest effective SQL*Plus script for finding the current value of a statistic is of the form:

```
COL NAME format a35
COL VALUE format 999,999,990
SELECT NAME, VALUE from V$SYSSTAT S
WHERE lower(NAME) LIKE lower('%&stat_name%')
/
```

Note: Two LOWER functions in the preceding query make it case insensitive and allow it to report data from the 11 statistics whose names start with "CPU" or "DBWR". No other upper-case characters appear in statistic names.

You can use the following query, for example, to report all statistics containing the word "get" in their name:

```
@STAT GET
```

It is preferable, however, to use mechanisms that record the change in the statistic(s) over a known period of time as described in the next section of this chapter.

Finding the Rate of Change of a Statistic

You can adapt the following script to show the rate of change for any statistic, latch, or event. For a given statistic, this script tells you the number of seconds between two checks of its value, and its rate of change.

```
set veri off
define secs=0
define value=0
col value format 99,999,999,990 new_value value
col secs format a10 new_value secs noprint
col delta format 9,999,990
col delta_time format 9,990
col rate format 999,990.0
col name format a30
select name,value, to_char(sysdate,’sssss’) secs,
    (value - &value) delta,
    (to_char(sysdate,’sssss’) - &secs) delta_time,
    (value - &value)/ (to_char(sysdate,’sssss’) - &secs) rate
    from v$sysstat
    where name = ’&&stat_name’
/
```

Note: Run this script at least twice, because the first time you run it, it initializes the SQL*Plus variables.

17

Diagnosing System Performance Problems

This chapter provides an overview of factors affecting performance in properly designed systems. Following the guidelines in this chapter cannot, however, compensate for poor design!

- Tuning Factors for Well Designed Existing Systems
- Insufficient CPU
- Insufficient Memory
- Insufficient I/O
- Network Constraints
- Software Constraints

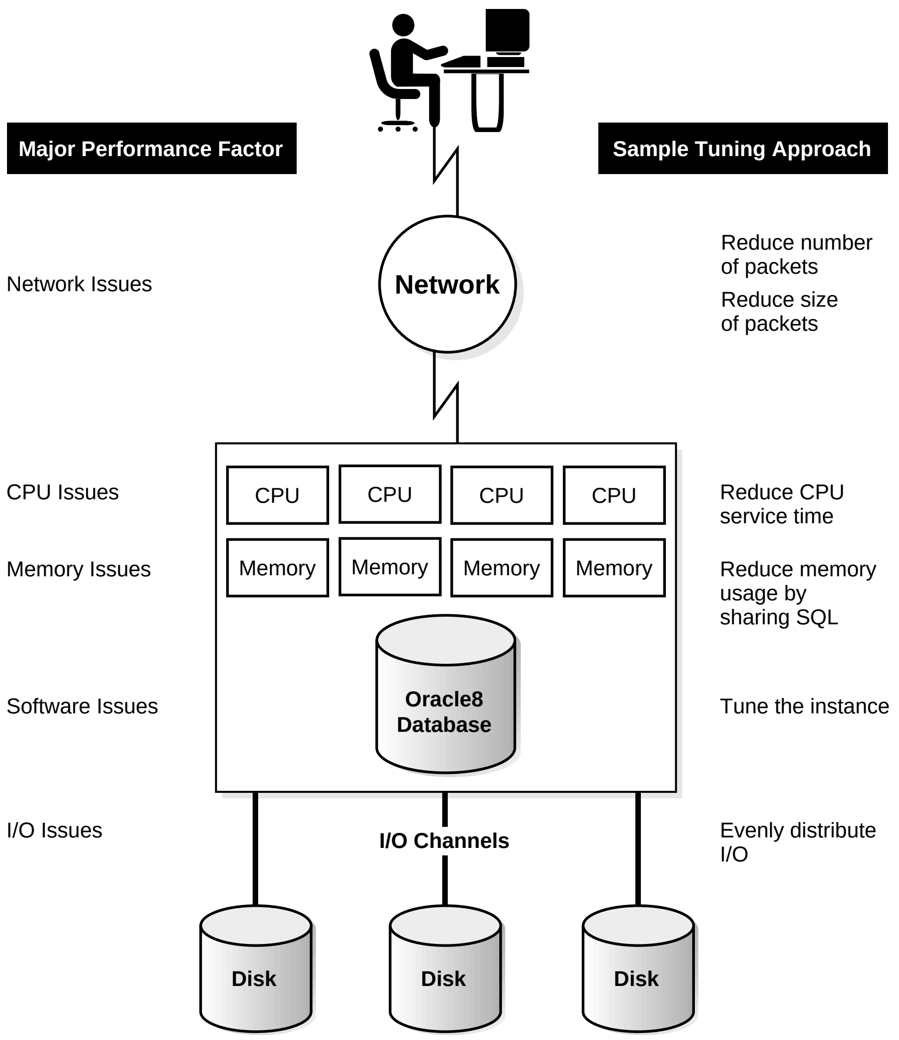
Later chapters discuss each of these factors in depth.

Tuning Factors for Well Designed Existing Systems

Figure 17–1 illustrates the factors involved in Oracle system performance for well designed applications.

Note: Tuning these factors is effective only *after* you have tuned the business process and the application, as described in Chapter 2, "Performance Tuning Methods".

Figure 17–1 Major Performance Factors in Well Designed Systems



Performance problems tend to be interconnected rather than isolated and unrelated. Table 17–1 identifies the key performance factors in existing systems as well as the areas in which symptoms may appear. For example, buffer cache problems may show up as CPU, memory, or I/O problems. Therefore, tuning the buffer cache CPU may improve I/O.

Table 17–1 Key to Tuning Areas for Existing Systems

ORACLE TUNING AREAS	LIMITING RESOURCES				
	CPU	Memory	I/O	Network	Software
Application					
Design / Architecture	X	X	X	X	X
DML SQL	X	X	X	X	X
Query SQL	X	X	X	X	X
Client / server Roundtrips	X			X	
Instance					
Buffer Cache	X	X	X		
Shared Pool	X	X			
Sort Area	X	X	X		
Physical Structure of Data / DB File I / O	X		X		
Log File I / O		X	X		
Archiver I / O	X		X		
Rollback Segments			X		X
Locking	X	X	X		X
Backups	X		X	X	X
Operating System					
Memory Management	X	X	X		
I / O Management	X	X	X		
Process Management	X	X			
Network Management		X		X	

Insufficient CPU

In a CPU-bound system, CPU resources might be completely allocated and service time could be excessive too. In this situation, you must improve your system’s processing ability. Alternatively, you could have too much idle time and the CPU might not be completely used up. In either case, you need to determine why so much time is spent waiting.

To determine why there is insufficient CPU, identify how your entire system is using CPU. Do not just rely on identifying how CPU is used by Oracle server processes. At the beginning of a workday, for example, the mail system may consume a large amount of available CPU while employees check their messages. Later in the day, the mail system may be much less of a bottleneck and its CPU use drops accordingly.

Workload is a very important factor when evaluating your system's level of CPU use. During peak workload hours, 90% CPU use with 10% idle and waiting time may be understandable and acceptable; 30% utilization at a time of low workload may also be understandable. However, if your system shows high utilization at normal workloads, there is no more room for a "peak workload". You have a CPU problem if idle time and time waiting for I/O are both close to zero, or less than 5%, at a normal or low workload.

See Also: Chapter 18, "Tuning CPU Resources".

Insufficient Memory

Sometimes a memory problem may be detected as an I/O problem. There are two types of memory requirements: Oracle and system. Oracle memory requirements affect the system requirements. Memory problems may be the cause of paging and swapping that occurs in the machine. So make sure your system does not start paging and swapping. The system should be able to run within the limitations set by internal memory.

System memory requirements for non-Oracle processes plus Oracle memory requirements should be equal to or less than the total available internal memory. To achieve this, reduce the size of some of the Oracle memory structures, such as the buffer cache, shared pool, or the redo log buffer. On the system level, you can reduce the number of processes and/or the amount of memory each process uses. You can also identify which processes are using the most memory. One way to reduce memory use is by sharing SQL.

See Also: Chapter 19, "Tuning Memory Allocation".

Insufficient I/O

Be sure to distribute I/O evenly across disks and channels. I/O resources include:

- Channel bandwidth: the number of I/O channels
- Device bandwidth: the number of disks

- Device latency: the time elapsed from the initiation of a request to the receipt of the request; latency is part of the "wait time"

I/O problems may result from hardware limitations. Your system needs enough disks and SCSI busses to support the transaction throughput you need. You can evaluate the configuration by calculating the quantity of messages all your disks and busses can potentially support, and comparing that to the number of messages required by your peak workload.

If the response time of an I/O becomes excessive, the most common problem is that wait time has increased (response time = service time + wait time). If wait time increases, it means there are too many I/O requests for this device. If service time increases, this normally means that the I/O requests are larger, so you write more bytes to disk.

The different background processes, such as DBWR, ARCH, and so on, perform different types of I/O, and each process has different I/O characteristics. Some processes read and write in the block size of the database, some read and write in larger chunks. If service time is too high, stripe the file across different devices.

Mirroring can also be a cause of I/O bottlenecks unless the data is mirrored to a destination database that has the same number of disks as the source database.

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

Network Constraints

Network constraints are similar to I/O constraints. You need to consider:

- Network bandwidth: Each transaction requires that a certain number of packets be sent over the network. If you know the number of packets required for one transaction, you can compare that to the bandwidth to determine whether your system is capable of supporting the desired workload.
- Message rates: You can reduce the number of packets on the network by batching them rather than sending many small packets.
- Transmission time

As the number of users increases and demand rises, the network can quietly become the bottleneck in an application. You may spend a lot of time waiting for network availability. Use available operating system tools to see how busy your network is.

See Also: Chapter 22, "Tuning Networks".

Software Constraints

Operating system software determines:

- The maximum number of processes you can support
- The maximum number of processes you can connect

Before you can tune Oracle effectively, ensure the operating system is performing optimally. Work closely with the hardware/software system administrators to ensure that Oracle is allocated the proper operating system resources.

Note: On NT systems there are no pre-set or configurable maximum numbers of processes that can be supported or connected.

See Also: Operating system tuning is different for every platform. Refer to your operating system hardware/software documentation as well as your Oracle operating system-specific documentation for more information. In addition, see Chapter 24, "Tuning the Operating System".

18

Tuning CPU Resources

This chapter describes how to solve CPU resource problems. Topics in this chapter include:

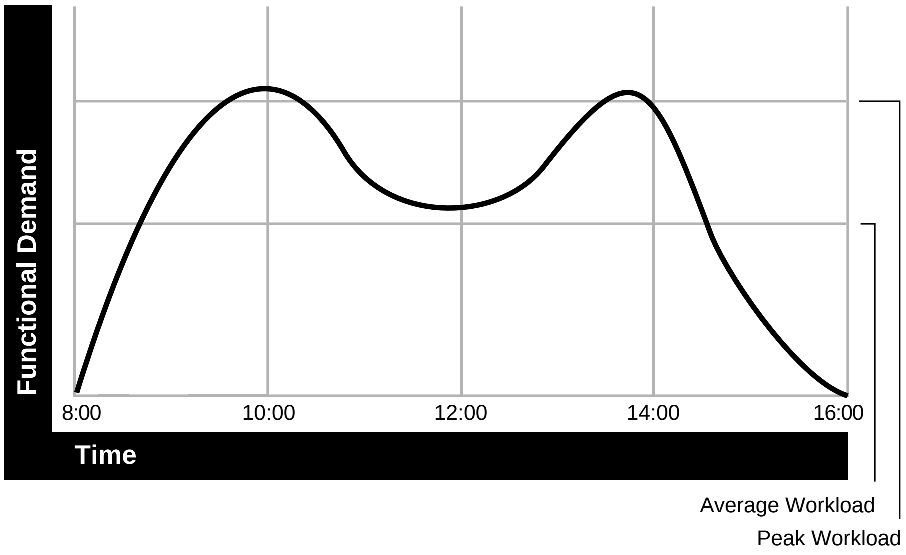
Understanding CPU Problems

To address CPU problems, first establish appropriate expectations for the amount of CPU resources your system should be using. Then distinguish 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 in three cases, when your system is:

- Idle
- At average workloads
- At peak workloads

Workload is an important factor when evaluating your system's level of CPU utilization. During peak workload hours, 90% CPU utilization with 10% idle and waiting time may be acceptable; 30% utilization at a time of low workload may also be understandable. However, if your system shows high utilization at normal workload, there is no room for a peak workload. For example, Figure 18–1 illustrates workload over time for an application having peak periods at 10:00 AM and 2:00 PM.

Figure 18–1 Average Workload and Peak Workload



This example application has 100 users working 8 hours a day, for a total of 800 hours per day. Each user entering one transaction every 5 minutes translates into 9,600 transactions daily. Over an 8-hour period, the system must support 1,200 transactions per hour, which is an average of 20 transactions per minute. If the demand rate were constant, you could build a system to meet this average workload.

However, usage patterns are not constant—and in this context, 20 transactions per minute can be understood as merely a minimum requirement. If the peak rate you need to achieve is 120 transactions per minute, you must configure a system that can support this peak workload.

For this example, assume that at peak workload Oracle can use 90% of the CPU resource. For a period of average workload, then, Oracle use no more than about 15% of the available CPU resource as illustrated in the following equation:

$$20\ tpm/120\ tpm\ *\ 90\% = 15\%$$

Where tpm is "transactions per minute".

If the system requires 50% of the CPU resource to achieve 20 tpm, then a problem exists: the system cannot achieve 120 transactions per minute using 90% of the CPU. However, if you tuned this system so it achieves 20 tpm using only 15% of the CPU, then, assuming linear scalability, the system might achieve 120 transactions per minute using 90% of the CPU resources.

As users are added to an application, the workload can rise to what had previously been peak levels. No further CPU capacity is then available for the new peak rate, which is actually higher than the previous.

Detecting and Solving CPU Problems

If you suspect a problem with CPU usage, check two areas:

- System CPU Utilization
- Oracle CPU Utilization

System CPU Utilization

Oracle statistics report CPU use by only Oracle sessions, whereas every process running on your system affects the available CPU resources. Effort spent tuning non-Oracle factors can thus result in improved Oracle performance.

Use operating system monitoring tools to determine what processes are running on the system as a whole. If the system is too heavily loaded, check the memory, I/O, and process management areas described later in this section.

Tools such as **sar -u** on many UNIX-based systems enable 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 NT, use Performance Monitor to examine CPU utilization. Performance Manager provides statistics on processor time, user time, privileged time, interrupt time, and DPC time. (NT Performance Monitor is not the same as Performance Manager, which is an Oracle Enterprise Manager tool.)

Note: This section describes how to check system CPU utilization on most UNIX-based and NT systems. For other platforms, please refer to your operating system documentation.

Memory Management

Check the following memory management areas:

Paging and Swapping Use tools such as **sar** or **vmstat** on UNIX or Performance Monitor on NT to investigate the cause of paging and swapping.

Oversize Page Tables On UNIX, if the processing space becomes too large, it may result in the page tables becoming too large. This is not an issue on NT.

I/O Management

Check the following I/O management issues:

Thrashing Ensure 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 in checking to be sure that it can run and to ensure all needed components are in the machine, the process may be using only 50% of the time allotted to actually perform work.

Client/Server Round Trips The latency of sending a message may result in CPU overload. An application often generates messages that need to be sent through the network over and over again, resulting in significant overhead before the message is actually sent. To alleviate this problem, batch the messages and perform the overhead only once or reduce the amount of work. For example, you can use array inserts, array fetches, and so on.

See Also: For more details on tuning I/O, please see Chapter 20, "Tuning I/O".

Process Management

Check the following process management issues:

Scheduling and Switching The operating system may spend excessive time scheduling and switching processes. Examine the way in which you are using the operating system because you could be using too many processes. On NT systems, do not overload your server with too many non-Oracle processes.

Context Switching Due to operating system specific characteristics, your system could be spending a lot of time in context switches. Context switching could be expensive, especially with a large SGA. Context switching is not an issue on NT which has only one process per instance; all threads share the same page table.

Programmers often create single-purpose processes, exit the process, and create a new one. Doing this re-creates and destroys the process each time. Such logic uses excessive amounts of CPU, especially with applications that have large SGAs. This is because you have 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, you may have page table entries for every 4K, and a page table entry may be 8 bytes. You could end up with (1G /4K) * 8B entries. This becomes expensive, because you have to continually make sure that the page table is loaded.

Parallel execution and multi-threaded server are areas of concern if MINSERVICE has been set too low (set to 10, for example, when you need 20). For an application that is performing small lookups, this may not be wise. In this situation, it becomes inefficient for the application and for the system as well.

Oracle CPU Utilization

This section explains how to examine the processes running in Oracle. Two dynamic performance views provide information on Oracle processes:

- V$SYSSTAT shows Oracle CPU usage for all sessions. The statistic "CPU Used" shows the aggregate CPU used by all sessions.
- V$SESSTAT shows Oracle CPU usage per session. You can use this view to determine which particular session is using the most CPU.

For example, if you have 8 CPUs, then for any given minute in real time, you have 8 minutes of CPU time available. On NT and UNIX, this can be either user time or time in system mode ("privileged" mode, in NT). If your process is not running, it is waiting. CPU time utilized by all systems may thus be greater than one minute per interval.

At any given moment you know how much time Oracle has used on the system. So if 8 minutes are available and Oracle uses 4 minutes of that time, then you know that 50% of all CPU time is used by Oracle. If your process is not consuming that time, then some other process is. You then need to identify the processes that are using CPU time. If you can, determine why the processes are using so much CPU time attempt to tune them.

The major areas to check for Oracle CPU utilization are:

- Reparsing SQL Statements
- Inefficient SQL Statements

- Read Consistency
- Scalability Limitations within the Application
- Latch Contention

This section describes each area and indicates the corresponding Oracle statistics to check.

Reparsing SQL Statements

When Oracle executes a SQL statement, it parses it to determine whether the syntax and its contents are correct. This process can consume significant overhead. Once parsed, Oracle does not parse the statement again unless the parsing information is aged from the memory cache and no longer available.

Ineffective memory sharing among SQL statements can result in reparsing. Use the following procedure to determine whether reparsing is occurring:

1. Begin by checking V$SYSSTAT to see if parsing in general is a problem:

```
SELECT * FROM V$SYSSTAT
WHERE NAME IN
('parse time cpu', 'parse time elapsed', 'parse count (hard)');
```

Where:

Response time	Service time + wait time, therefore response time = elapsed time.
Service time	CPU time, therefore elapsed time - CPU time = wait time.

In this way, you can detect the general response time on parsing. The more your application is parsing, the more contention exists and the more time your system spends waiting. Note the following:

Wait time / parse count	Average wait time per parse.

Note: The average wait time should be extremely low, approaching zero. (V$SYSSTAT also shows the average wait time per parse.)

2. Query V$SQLAREA to find frequently reparsed statements:

```
SELECT SQL_TEXT, PARSE_CALLS, EXECUTIONS FROM V$SQLAREA
ORDER BY PARSE_CALLS;
```

Tune the statements with the higher numbers of parse calls.

3. You have the following three options for tuning them:

- Rewrite the application so statements do not continually reparse.
- Reduce parsing by using the initialization parameter SESSION_CACHED_CURSORS.
- If the parse count is small, the execute count is small, and the SQL statements are very similar except for the WHERE clause, you may find that hard coded values are being used instead of bind variables. Use bind variables to reduce parsing.

Inefficient SQL Statements

Inefficient SQL statements can consume large amounts of CPU resources. To detect such statements, enter the following query. You may be able to reduce CPU usage by tuning SQL statements that have a high number of buffer gets.

```
SELECT BUFFER_GETS, EXECUTIONS, SQL_TEXT FROM V$SQLAREA;
```

See Also: "Approaches to SQL Statement Tuning" on page 4-6.

Read Consistency

Your system may spend excessive time rolling back changes to blocks in order to maintain a consistent view. Consider the following scenarios:

- If there are many small transactions and an active long-running query is running in the background on the same table where the inserts are happening, the query may have to roll back many changes.
- If the number of rollback segments is too small, your system could also be spending a lot of time rolling back the transaction table. Your query may have

started long ago; because the number of rollback segments and transaction tables is very small, your system frequently needs to reuse transaction slots.

A solution is to make more rollback segments, or to increase the commit rate. For example, if you batch ten transactions and commit them once, you reduce the number of transactions by a factor of ten.

- If your system must scan too many buffers in the foreground to find a free buffer, it wastes CPU resources. To alleviate this problem, tune the DBW*n* process(es) to write more frequently.

  You can also increase the size of the buffer cache to enable the database writer process(es) to keep up. To find the average number of buffers the system scans at the end of the least recently used list (LRU) to find a free buffer, use the following formula:

$$\frac{1 + \textit{value of "free buffers inspected"}}{\textit{"free buffers inspected"}} = \textit{avg. buffers scanned}$$

  On average, you would expect to see 1 or 2 buffers scanned. If more than this number are being scanned, increase the size of the buffer cache or tune the DBW*n* process(es).

Use the following formula to find the number of buffers that were dirty at the end of the LRU:

$$\frac{\textit{"dirty buffers inspected"}}{\textit{"free buffers inspected"}} = \textit{dirty buffers}$$

If many dirty buffers exist, it could mean that the DBW*n* process(es) cannot keep up. Again, increase the buffer cache size or tune the DBW*n* process.

Scalability Limitations Within the Application

In most of this CPU tuning discussion, we assume you can achieve linear scalability, but this is never actually the case. How flat or nonlinear the scalability is indicates how far away from optimal performance your system is. Problems in your application might be adversely affecting scalability. Examples of this include too many indexes, right-hand index problems, too much data in the blocks, or not properly partitioning the data. These types of contention problems waste CPU cycles and prevent the application from attaining linear scalability.

Latch Contention

Latch contention is a symptom of CPU problems; it is not normally a cause. To resolve it, you must locate the latch contention within your application, identify its cause, and determine which part of your application is poorly written.

In some cases, the spin count may be set too high. It’s also possible that one process may be holding a latch that another process is attempting to secure. The process attempting to secure the latch may be endlessly spinning. After a while, this process may go to sleep and later resume processing and repeat its ineffectual spinning. To resolve this:

- Check the Oracle latch statistics. The "latch free" event in V$SYSTEM_EVENT shows how long processes have been waiting for latches. If there is no latch contention, this statistic will not appear.

  If there is a lot of contention, it may be better for a process to go to sleep at once when it cannot obtain a latch, rather than use CPU time by spinning.

- Look for the ratio of CPUs to processes. If there are large numbers of both, then many processes can run. But if a single process is holding a latch on a system with ten CPUs, reschedule that process so it is not running. But ten other processes may run ineffectively trying to secure the same latch. This situation wastes, in parallel, some CPU resource.

- Check V$LATCH_MISSES, which indicates where in the Oracle code most contention occurs.

Solving CPU Problems by Changing System Architectures

If you have maximized the CPU power on your system and have exhausted all means of tuning your system's CPU use, consider redesigning your system on another architecture. Moving to a different architecture might improve CPU use. This section describes architectures you might consider using, such as:

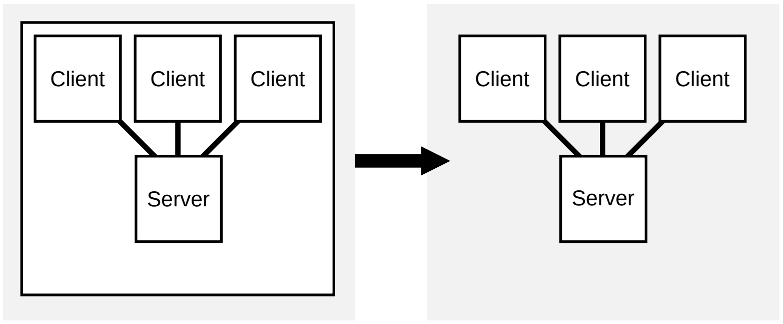
- Single Tier to Two-Tier
- Multi-Tier: Using Smaller Client Machines
- Two-Tier to Three-Tier: Using a Transaction Processing Monitor
- Three-Tier: Using Multiple TP Monitors
- Oracle Parallel Server

Note: If you are running a multi-tier system, check all levels for CPU utilization. For example, on a three-tier system, your server might be mostly idle while your second tier is completely busy. To resolve this, tune the second tier rather than the server or the third tier. In a multi-tier system, it is usually not the server that has a performance problem: it is usually the clients and the middle tier.

Single Tier to Two-Tier

Consider whether changing from several clients with one server, all running on a single machine (single tier), to a two-tier client/server configuration would relieve CPU problems.

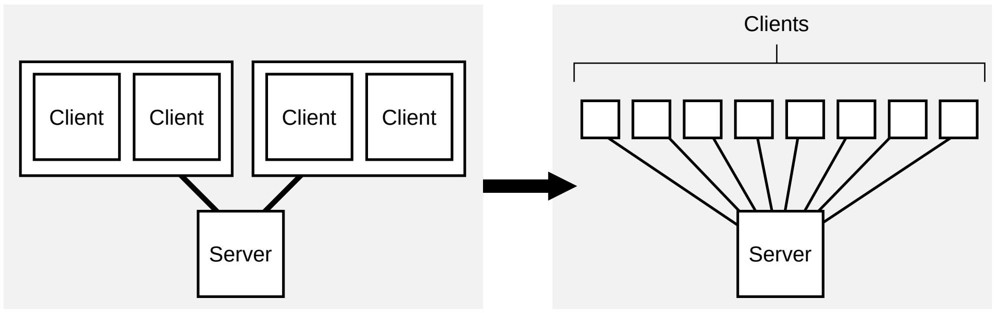
Figure 18–2 Single Tier to Two-Tier



Multi-Tier: Using Smaller Client Machines

Consider whether using smaller clients improves CPU usage rather than using multiple clients on larger machines. This strategy may be helpful with either two-tier or three-tier configurations.

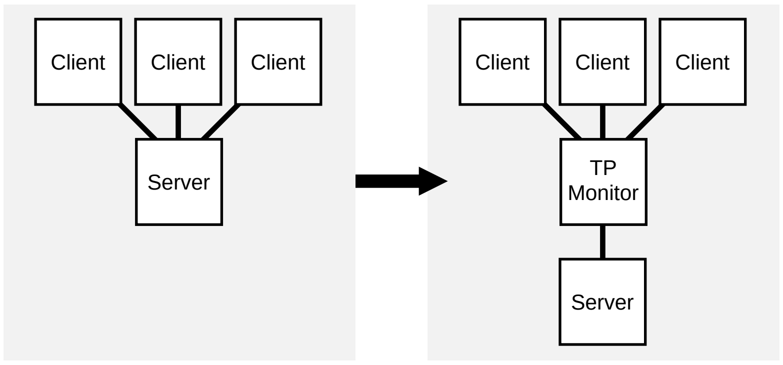
Figure 18–3 Multi-Tier Using Smaller Clients



Two-Tier to Three-Tier: Using a Transaction Processing Monitor

If your system runs with multiple layers, consider whether moving from a two-tier to three-tier configuration and introducing a transaction processing monitor might be a good solution.

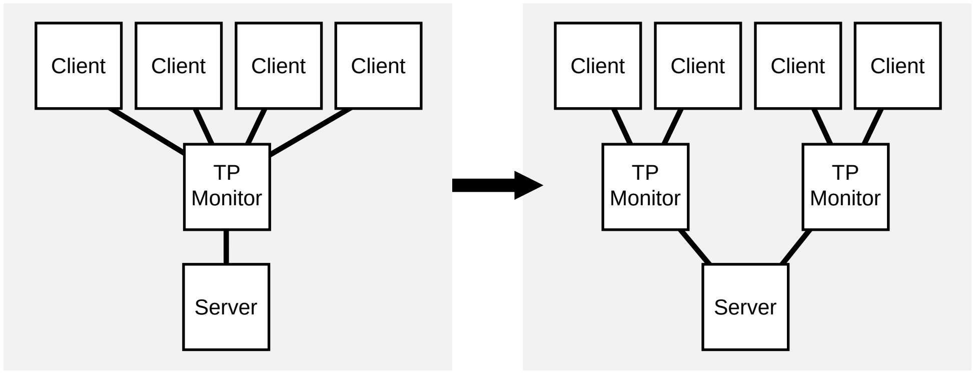
Figure 18–4 Two-Tier to Three-Tier



Three-Tier: Using Multiple TP Monitors

Consider using multiple transaction processing monitors.

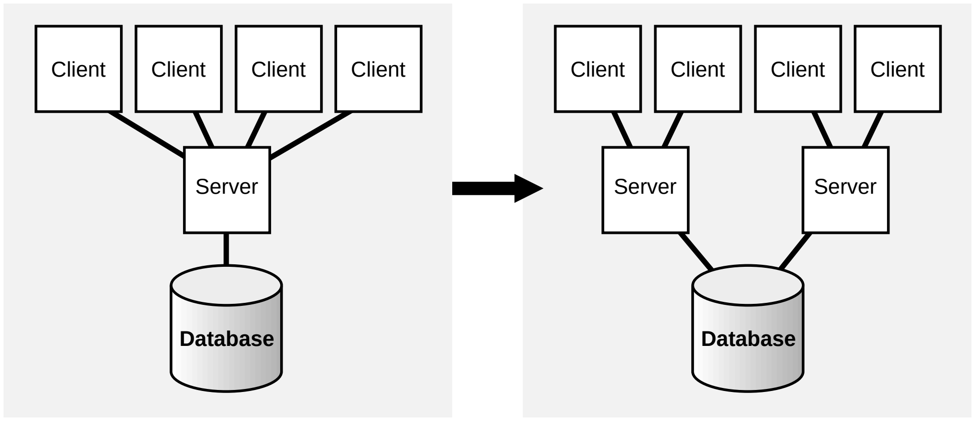
Figure 18–5 Three-Tier with Multiple TP Monitors



Oracle Parallel Server

Consider whether incorporating Oracle Parallel Server would solve your CPU problems.

Figure 18–6 Oracle Parallel Server



19

Tuning Memory Allocation

This chapter explains how to allocate memory to database structures. Proper sizing of these structures greatly improves database performance. The following topics are covered:

Understanding Memory Allocation Issues

Oracle stores information in memory and on disk. Memory access is much faster than disk access, so it is better for data requests to be satisfied by access to memory rather than access to disk. For best performance, store as much data as possible in memory rather than on disk. However, memory resources on your operating system are likely to be limited. Tuning memory allocation involves distributing available memory to Oracle memory structures.

Oracle's memory requirements depend on your application. Therefore, tune memory allocation after tuning your application and SQL statements. If you allocate memory before tuning your application and SQL statements, you may need to resize some Oracle memory structures to meet the needs of your modified statements and application.

Also tune memory allocation before you tune I/O. Allocating memory establishes the amount of I/O necessary for Oracle to operate. This chapter shows you how to allocate memory to perform as little I/O as possible.

The following terms are used in this discussion:

block	A unit of data transfer between main memory and disk. Many blocks from one section of memory address space form a segment.
buffer	A main memory address in which the buffer manager caches currently and recently used data read from disk. Over time a buffer may hold different blocks. When a new block is needed, the buffer manager may discard an old block and replace it with a new one.
buffer pool	A collection of buffers.
cache or buffer cache	All buffers and buffer pools.
segment	A segment is a set of extents allocated for a specific type of database object such as a table, index, or cluster.

See Also: Chapter 20, "Tuning I/O" shows you how to perform I/O as efficiently as possible.

Detecting Memory Allocation Problems

When you use operating system tools such as **ps -efl** or **ps - aux** on UNIX to examine the size of Oracle processes, you may notice that the processes seem large. To interpret the statistics shown, determine how much of the process size is attributable to shared memory, heap, and executable stack, and how much is the actual amount of memory the given process consumes.

The SZ statistic is given in units of page size (normally 4KB), and normally includes the shared overhead. To calculate the private, or per-process memory usage, subtract shared memory and executable stack figures from the value of SZ. For example:

SZ	+20,000
minus SHM	- 15,000
minus EXECUTABLE	- 1,000
actual per-process memory	4,000

In this example, the individual process consumes only 4,000 pages; the other 16,000 pages are shared by all processes.

> **See Also:** *Oracle for UNIX Performance Tuning Tips*, or your operating system documentation.

Solving Memory Allocation Problems

The rest of this chapter explains how to tune memory allocation. For best results, resolve memory issues in the order presented here:

1. Tuning Operating System Memory Requirements
2. Tuning the Redo Log Buffer
3. Tuning Private SQL and PL/SQL Areas
4. Tuning the Shared Pool
5. Tuning the Buffer Cache
6. Tuning Multiple Buffer Pools
7. Tuning Sort Areas
8. Reallocating Memory
9. Reducing Total Memory Usage

Tuning Operating System Memory Requirements

Begin tuning memory allocation by tuning your operating system with these goals:

- Reducing Paging and Swapping
- Fitting the System Global Area into Main Memory
- Allocating Adequate Memory to Individual Users

These goals apply in general to most operating systems, but the details of operating system tuning vary.

> **See Also:** Refer to your operating system hardware and software documentation as well as your Oracle operating system-specific documentation for more information on tuning operating system memory usage.

Reducing Paging and Swapping

Your operating system may store information in these places:

- Real memory
- Virtual memory
- Expanded storage
- Disk

The operating system may also move information from one storage location to another. This process is known as "paging" or "swapping". Many operating systems page and swap to accommodate large amounts of information that do not fit into real memory. However, excessive paging or swapping can reduce the performance of many operating systems.

Monitor your operating system behavior with operating system utilities. Excessive paging or swapping indicates that new information is often being moved into memory. In this case, your system's total memory may not be large enough to hold everything for which you have allocated memory. Either increase the total memory on your system or decrease the amount of memory allocated.

> **See Also:** "Oversubscribing with Attention to Paging" on page 27-5.

Fitting the System Global Area into Main Memory

Because the purpose of the System Global Area (SGA) is to store data in memory for fast access, the SGA should always be within main memory. If pages of the SGA are swapped to disk, its data is no longer quickly accessible. On most operating systems, the disadvantage of excessive paging significantly outweighs the advantage of a large SGA.

Although it is best to keep the entire SGA in memory, the contents of the SGA will be split logically between "hot" and "cold" parts. The hot parts are always in memory because they are always being referenced. Some cold parts may be paged out and a performance penalty may result from bringing them back in. A performance problem likely occurs, however, when the hot part of the SGA cannot remain in memory.

Data is swapped to disk because it is not being referenced. You can cause Oracle to read the entire SGA into memory when you start your instance by setting the value of the initialization parameter PRE_PAGE_SGA to YES. Operating system page table entries are then pre-built for each page of the SGA. This setting may increase the amount of time necessary for instance startup, but it is likely to decrease the amount of time necessary for Oracle to reach its full performance capacity after startup.

Note: This setting does not prevent your operating system from paging or swapping the SGA after it is initially read into memory.)

PRE_PAGE_SGA may increase the process startup duration, because every process that starts must attach itself to the SGA. The cost of this strategy is fixed, however; you may simply determine that 20,000 pages must be touched every time a process starts. This approach may be useful with some applications, but not with all applications. Overhead may be significant if your system frequently creates and destroys processes by, for example, continually logging on and logging off.

The advantage that PRE_PAGE_SGA can afford depends on page size. For example, if the SGA is 80MB in size, and the page size is 4KB, then 20,000 pages must be touched to refresh the SGA (80,000 / 4 = 20,000).

If the system permits you to set a 4MB page size, then only 20 pages must be touched to refresh the SGA (80,000 / 4,000 = 20). The page size is operating-system specific and generally cannot be changed. Some operating systems, however, have a special implementation for shared memory whereby you can change the page size.

You can see how much memory is allocated to the SGA and each of its internal structures by issuing this SQL statement:

```
SHOW SGA
```

The output of this statement might look like this:

```
Total System Global Area                              18847360 bytes
Fixed Size                                               63104 bytes
Variable Size                                         14155776 bytes
Database Buffers                                       4096000 bytes
Redo Buffers                                            532480 bytes
```

Some IBM mainframe computer operating systems have expanded storage or special memory in addition to main memory to which paging can be performed very quickly. These operating systems may be able to page data between main memory and expanded storage faster than Oracle can read and write data between the SGA and disk. For this reason, allowing a larger SGA to be swapped may lead to better performance than ensuring that a smaller SGA remains in main memory. If your operating system has expanded storage, take advantage of it by allocating a larger SGA despite the resulting paging.

Allocating Adequate Memory to Individual Users

On some operating systems, you may have control over the amount of physical memory allocated to each user. Be sure all users are allocated enough memory to accommodate the resources they need to use their application with Oracle.

Depending on your operating system, these resources may include:

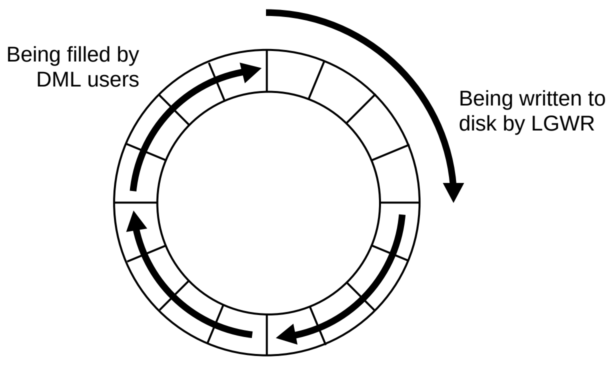
- The Oracle executable image
- The SGA
- Oracle application tools
- Application-specific data

On some operating systems, Oracle software can be installed so that a single executable image can be shared by many users. By sharing executable images among users, you can reduce the amount of memory required by each user.

Tuning the Redo Log Buffer

The LOG_BUFFER parameter reserves space for the redo log buffer that is fixed in size. On machines with fast processors and relatively slow disks, the processors may be filling the rest of the buffer in the time it takes the redo log writer to move a portion of the buffer to disk. The log writer process (LGWR) always starts when the buffer begins to fill. For this reason, a larger buffer makes it less likely that new entries collide with the part of the buffer still being written.

Figure 19–1 Redo Log Buffer



The log buffer is normally small compared with the total SGA size, and a modest increase can significantly enhance throughput. A key ratio is the space request ratio: redo log space requests / redo entries. If this ratio is greater than 1:5000, increase the size of the redo log buffer until the space request ratio stops falling.

Tuning Private SQL and PL/SQL Areas

This section explains how to tune private SQL and PL/SQL areas.

- Identifying Unnecessary Parse Calls
- Reducing Unnecessary Parse Calls

A trade-off exists between memory and reparsing. With significant amounts of reparsing, less memory is needed. If you reduce reparsing by creating more SQL statements, then client memory requirements increase. This is due to an increase in the number of open cursors.

Tuning private SQL areas entails identifying unnecessary parse calls made by your application and then reducing them. To reduce parse calls, you may have to increase the number of private SQL areas that your application can have allocated at once. Throughout this section, information about private SQL areas and SQL statements also applies to private PL/SQL areas and PL/SQL blocks.

Identifying Unnecessary Parse Calls

This section describes three techniques for identifying unnecessary parse calls.

Technique 1

One way to identify unnecessary parse calls is to run your application with the SQL trace facility enabled. For each SQL statement in the trace output, the "count" statistic for the Parse step tells you how many times your application makes a parse call for the statement. This statistic includes parse calls satisfied by access to the library cache as well as parse calls resulting in actually parsing the statement.

Note: This statistic does not include implicit parsing that occurs when an application executes a statement whose shared SQL area is no longer in the library cache. For information on detecting implicit parsing, see "Examining Library Cache Activity" on page 19-13.

If the "count" value for the Parse step is near the "count" value for the Execute step for a statement, your application may be deliberately making a parse call each time it executes the statement. Try to reduce these parse calls through your application tool.

Technique 2

Another way to identify unnecessary parse calls is to check the V$SQLAREA view. Enter the following query:

```
SELECT SQL_TEXT, PARSE_CALLS, EXECUTIONS
FROM V$SQLAREA
```

When the PARSE_CALLS value is close to the EXECUTION value for a given statement, you may be continually reparsing that statement.

Technique 3

You can also identify unnecessary parse calls by identifying the session in which they occur. It may be that particular batch programs or certain types of applications do most of the reparsing. To do this, execute the following query:

```
SELECT * FROM V$STATNAME
```

```
WHERE NAME IN ('PARSE_COUNT (HARD)','EXECUTE_COUNT')
```

Oracle responds with something similar to:

```
STATISTIC#,   NAME
------------  ---------
100           PARSE_COUNT
90            EXECUTE_COUNT
```

Then run a query similar to the following:

```
SELECT * FROM V$SESSTAT
  WHERE STATISTICS# IN (90,100)
  ORDER BY VALUE, SID;
```

The result is a list of all sessions and the amount of reparsing they do. For each system identifier (sid), go to V$SESSION to find the name of the program that causes the reparsing.

Reducing Unnecessary Parse Calls

Depending on the Oracle application tool you are using, you may be able to control how frequently your application performs parse calls and allocates and deallocates private SQL areas. Whether your application reuses private SQL areas for multiple SQL statements determines how many parse calls your application performs and how many private SQL areas the application requires.

In general, an application that reuses private SQL areas for multiple SQL statements does not need as many private SQL areas as an application that does not reuse private SQL areas. However, an application that reuses private SQL areas must perform more parse calls, because the application must make a new parse call whenever an existing private SQL area is reused for a new SQL statement.

Be sure your application can open enough private SQL areas to accommodate all your SQL statements. If you allocate more private SQL areas, you may need to increase the limit on the number of cursors permitted for a session. You can increase this limit by increasing the value of the initialization parameter OPEN_CURSORS. The maximum value for this parameter depends on your operating system. The minimum value is 5.

The ways in which you control parse calls and allocation and deallocation of private SQL areas depends on your Oracle application tool. The following sections introduce the methods used for some tools. These methods apply only to private SQL areas and not to shared SQL areas.

Reducing Parse Calls with the Oracle Precompilers

When using the Oracle precompilers, you can control private SQL areas and parse calls by setting three options. In Oracle mode, the options and their defaults are as follows:

- HOLD_CURSOR = yes
- RELEASE_CURSOR = no
- MAXOPENCURSORS = *desired value*

Oracle recommends that you *not* use ANSI mode, in which the values of HOLD_CURSOR and RELEASE_CURSOR are switched.

The precompiler options can be specified in two ways:

- On the precompiler command line
- Within the precompiler program

With these options, you can employ different strategies for managing private SQL areas during the course of the program.

> **See Also:** *Pro*C/C++ Precompiler Programmer's Guide* for more information on these calls.

Reducing Parse Calls with Oracle Forms

With Oracle Forms, you also have some control over whether your application reuses private SQL areas. You can exercise this control in three places:

- At the trigger level
- At the form level
- At run time

> **See Also:** For more information on the reuse of private SQL areas by Oracle Forms, see the *Oracle Forms Reference* manual.

Tuning the Shared Pool

This section explains how to allocate memory for key memory structures of the shared pool. Structures are listed in order of importance for tuning.

- Tuning the Library Cache
- Tuning the Data Dictionary Cache

- Tuning the Large Pool and Shared Pool for the Multi-threaded Server Architecture
- Tuning Reserved Space from the Shared Pool

> **Note:** If you are using a reserved size for the shared pool, refer to "SHARED_POOL_SIZE Too Small" on page 19-25.

The algorithm Oracle uses to manage data in the shared pool tends to hold dictionary cache data in memory longer than library cache data. Therefore, tuning the library cache to an acceptable cache hit ratio often ensures that the data dictionary cache hit ratio is also acceptable. Allocating space in the shared pool for session information is necessary only if you are using MTS (Multi-threaded Server) architecture.

In the shared pool, some of the caches are dynamic—their sizes automatically increase or decrease as needed. These dynamic caches include the library cache and the data dictionary cache. Objects are aged out of these caches if the shared pool runs out of room. For this reason you may need to increase the shared pool size if the frequently used set of data does not fit within it. A cache miss on the data dictionary cache or library cache is more expensive than a miss on the buffer cache. For this reason, allocate sufficient memory to the shared pool before allocating to the buffer cache.

For most applications, shared pool size is critical to Oracle performance. (Shared pool size is less important only for applications that issue a very limited number of discrete SQL statements.) The shared pool holds both the data dictionary cache and the fully parsed or compiled representations of PL/SQL blocks and SQL statements. PL/SQL blocks include procedures, functions, packages, triggers and any anonymous PL/SQL blocks submitted by client programs.

If the shared pool is too small, the server must dedicate resources to managing the limited amount of available space. This consumes CPU resources and causes contention because Oracle imposes restrictions on the parallel management of the various caches. The more you use triggers and stored procedures, the larger the shared pool must be. It may even reach a size measured in hundreds of megabytes.

Because it is better to measure statistics during a confined period rather than from startup, you can determine the library cache and row cache (data dictionary cache) hit ratios from the following queries. The results show the miss rates for the library cache and row cache. In general, the number of reparses reflects the library cache. If the ratios are close to 1, you do not need to increase the pool size.

```
SELECT (SUM(PINS - RELOADS)) / SUM(PINS) "LIB CACHE"
 FROM V$LIBRARYCACHE;

SELECT (SUM(GETS - GETMISSES - USAGE - FIXED)) / SUM(GETS) "ROW CACHE"
 FROM V$ROWCACHE;
```

The amount of free memory in the shared pool is reported in V$SGASTAT. Report the current value from this view using the query:

```
SELECT * FROM V$SGASTAT WHERE NAME = 'FREE MEMORY';
```

If there is always free memory available within the shared pool, then increasing the size of the pool offers little or no benefit. However, just because the shared pool is full does not necessarily mean there is a problem.

Once an entry has been loaded into the shared pool it cannot be moved. As more entries are loaded, the free memory becomes discontiguous, and the shared pool may become fragmented.

You can use the PL/SQL package DBMS_SHARED_POOL, located in **dbmspool.sql,** to manage the shared pool. The comments in the code describe how to use the procedures within the package.

> **See Also:** For more information about DBMS_SHARED_POOL, see the *Oracle8i Supplied Packages Reference.*

Loading PL/SQL Objects into the Shared Pool

Oracle loads objects into the shared pool using "pages" that are 4KB in size. These pages load chunks of segmented PL/SQL code. The pages do not need to be contiguous. Therefore, Oracle does not need to allocate large sections of contiguous memory for loading objects into the shared pool. This reduces the need for contiguous memory and improves performance. However, Oracle loads all of a package if any part of the package is called.

Depending on user needs, it may or may not be prudent to pin packages in the shared pool. Nonetheless, Oracle recommends pinning, especially for frequently used application objects.

Library Cache and Row Cache Hit Ratios

Library cache and row cache hit ratios are important. If free memory is near zero and either the library cache hit ratio or the row cache hit ratio is less than 0.95, increase the size of the shared pool until the ratios stop improving.

Tuning the Library Cache

This section describes how to tune the library cache hit ratio.

The library cache holds executable forms of SQL cursors, PL/SQL programs, and JAVA classes. It also caches descriptive information, or metadata, about schema objects. Oracle uses this metadata when parsing SQL cursors or during the compilation of PL/SQL programs. The latter type of memory is seldom a concern for performance, so this section focuses on tuning as it relates to cursors, PL/SQL programs, and JAVA classes. These are collectively referred to as "application logic".

Examining Library Cache Activity

Library cache misses can occur on either the parse or the execute step in the processing of a SQL statement.

Parse If an application makes a parse call for a SQL statement and the parsed representation of the statement does not already exist in a shared SQL area in the library cache, Oracle parses the statement and allocates a shared SQL area. You may be able to reduce library cache misses on parse calls by ensuring that SQL statements can share a shared SQL area whenever possible.

Execute If an application makes an execute call for a SQL statement and the shared SQL area containing the parsed representation of the statement has been deallocated from the library cache to make room for another statement, Oracle implicitly reparses the statement, allocates a new shared SQL area for it, and executes it. You may be able to reduce library cache misses on execution calls by allocating more memory to the library cache.

To determine whether misses on the library cache are affecting the performance of Oracle, query the dynamic performance table V$LIBRARYCACHE.

The V$LIBRARYCACHE View You can monitor statistics reflecting library cache activity by examining the dynamic performance view V$LIBRARYCACHE. These statistics reflect all library cache activity since the most recent instance startup. By default, this view is available only to the user SYS and to users granted SELECT ANY TABLE system privilege, such as SYSTEM.

Each row in this view contains statistics for one type of item kept in the library cache. The item described by each row is identified by the value of the NAMESPACE column. Rows of the table with the following NAMESPACE values reflect library cache activity for SQL statements and PL/SQL blocks:

- SQL AREA

- TABLE/PROCEDURE
- BODY
- TRIGGER

Rows with other NAMESPACE values reflect library cache activity for object definitions that Oracle uses for dependency maintenance.

These columns of the V$LIBRARYCACHE table reflect library cache misses on execution calls:

PINS	This column shows the number of times an item in the library cache was executed.
RELOADS	This column shows the number of library cache misses on execution steps.

Querying the V$LIBRARYCACHE Table Monitor the statistics in the V$LIBRARYCACHE table over a period of time with this query:

```
SELECT SUM(PINS) "EXECUTIONS",
   SUM(RELOADS) "CACHE MISSES WHILE EXECUTING"
   FROM V$LIBRARYCACHE;
```

The output of this query might look like this:

```
EXECUTIONS CACHE MISSES WHILE EXECUTING
---------- ----------------------------
  320871                  549
```

Interpreting the V$LIBRARYCACHE Table Examining the data returned by the sample query leads to these observations:

- The sum of the EXECUTIONS column indicates that SQL statements, PL/SQL blocks, and object definitions were accessed for execution a total of 320,871 times.
- The sum of the CACHE MISSES WHILE EXECUTING column indicates that 549 of those executions resulted in library cache misses causing Oracle to implicitly reparse a statement or block or reload an object definition because it aged out of the library cache.
- The ratio of the total misses to total executions is about 0.17%. This value means that only 0.17% of executions resulted in reparsing.

Total misses should be near 0. If the ratio of misses to executions is more than 1%, try to reduce the library cache misses through the means discussed in the next section.

Reducing Library Cache Misses

You can reduce library cache misses by:

- Allocating additional memory to the library cache
- Writing identical SQL statements whenever possible

Allocating Additional Memory for the Library Cache You may be able to reduce library cache misses on execution calls by allocating additional memory for the library cache. To ensure that shared SQL areas remain in the cache once their SQL statements are parsed, increase the amount of memory available to the library cache until the V$LIBRARYCACHE.RELOADS value is near 0. To increase the amount of memory available to the library cache, increase the value of the initialization parameter SHARED_POOL_SIZE. The maximum value for this parameter depends on your operating system. This measure will reduce implicit reparsing of SQL statements and PL/SQL blocks on execution.

To take advantage of additional memory available for shared SQL areas, you may also need to increase the number of cursors permitted for a session. You can do this by increasing the value of the initialization parameter OPEN_CURSORS.

Be careful not to induce paging and swapping by allocating too much memory for the library cache. The benefits of a library cache large enough to avoid cache misses can be partially offset by reading shared SQL areas into memory from disk whenever you need to access them.

See Also: "SHARED_POOL_SIZE Too Small" on page 19-25.

Writing Identical SQL Statements: Criteria You may be able to reduce library cache misses on parse calls by ensuring that SQL statements and PL/SQL blocks use a shared SQL area whenever possible. Two separate occurrences of a SQL statement or PL/SQL block can use a shared SQL area if they are identical according to these criteria:

- The text of the SQL statements or PL/SQL blocks must be identical, character for character, including spaces and case. For example, these statements cannot use the same shared SQL area:

  ```
  SELECT * FROM EMP;
  SELECT *  FROM EMP;
  ```

These statements cannot use the same shared SQL area:

```
SELECT * FROM EMP;
SELECT * FROM EMP;
```

- References to schema objects in the SQL statements or PL/SQL blocks must resolve to the same object in the same schema.

  For example, if the schemas of the users BOB and ED both contain an EMP table and both users issue the following statement, their statements cannot use the same shared SQL area:

  ```
  SELECT * FROM EMP;
  SELECT * FROM EMP;
  ```

  If both statements query the same table and qualify the table with the schema, as in the following statement, then they can use the same shared SQL area:

  ```
  SELECT * FROM BOB.EMP;
  ```

- Bind variables in the SQL statements must match in name and datatype. For example, these statements cannot use the same shared SQL area:

  ```
  SELECT * FROM EMP WHERE DEPTNO = :DEPARTMENT_NO;
  SELECT * FROM EMP WHERE DEPTNO = :D_NO;
  ```

- The SQL statements must be optimized using the same optimization approach and, in the case of the cost-based approach, the same optimization goal. For information on optimization approach and goal, see "Choosing a Goal for the Cost-based Approach" on page 7-3.

Writing Identical SQL Statements: Strategies Shared SQL areas are most useful for reducing library cache misses for multiple users running the same application. Discuss these criteria with the developers of such applications and agree on strategies to ensure that the SQL statements and PL/SQL blocks of an application can use the same shared SQL areas:

- Use bind variables rather than explicitly specified constants in your statements whenever possible.

  For example, the following two statements cannot use the same shared area because they do not match character for character:

  ```
  SELECT ENAME, EMPNO FROM EMP WHERE DEPTNO = 10;
  SELECT ENAME, EMPNO FROM EMP WHERE DEPTNO = 20;
  ```

You can accomplish the goals of these statements by using the following statement that contains a bind variable, binding 10 for one occurrence of the statement and 20 for the other:

```
SELECT ENAME, EMPNO FROM EMP WHERE DEPTNO = :DEPARTMENT_NO;
```

The two occurrences of the statement can then use the same shared SQL area.

- Be sure that users of the application do not change the optimization approach and goal for their individual sessions.
- You can also increase the likelihood that SQL statements issued by different applications can share SQL areas by establishing these policies among the developers of the applications:
  - Standardize naming conventions for bind variables and spacing conventions for SQL statements and PL/SQL blocks.
  - Use stored procedures whenever possible. Multiple users issuing the same stored procedure automatically use the same shared PL/SQL area. Since stored procedures are stored in a parsed form, they eliminate run-time parsing altogether.

Using CURSOR_SPACE_FOR_TIME to Speed Access to Shared SQL Areas

If you have no library cache misses, you may still be able to accelerate execution calls by setting the value of the initialization parameter CURSOR_SPACE_FOR_TIME. This parameter specifies whether a shared SQL area can be deallocated from the library cache to make room for a new SQL statement. CURSOR_SPACE_FOR_TIME has the following values meanings:

- If this parameter is set to FALSE (the default), a shared SQL area can be deallocated from the library cache regardless of whether application cursors associated with its SQL statement are open. In this case, Oracle must verify that a shared SQL area containing the SQL statement is in the library cache.
- If this parameter is set to TRUE, a shared SQL area can be deallocated only when all application cursors associated with its statement are closed. In this case, Oracle need not verify that a shared SQL area is in the cache, because the shared SQL area can never be deallocated while an application cursor associated with it is open.

Setting the value of the parameter to TRUE saves Oracle a small amount of time and may slightly improve the performance of execution calls. This value also prevents the deallocation of private SQL areas until associated application cursors are closed.

Do not set the value of CURSOR_SPACE_FOR_TIME to TRUE if you have found library cache misses on execution calls. Such library cache misses indicate that the shared pool is not large enough to hold the shared SQL areas of all concurrently open cursors. If the value is TRUE and the shared pool has no space for a new SQL statement, the statement cannot be parsed and Oracle returns an error saying that there is no more shared memory. If the value is FALSE and there is no space for a new statement, Oracle deallocates an existing shared SQL area. Although deallocating a shared SQL area results in a library cache miss later, it is preferable to an error halting your application because a SQL statement cannot be parsed.

Do not set the value of CURSOR_SPACE_FOR_TIME to TRUE if the amount of memory available to each user for private SQL areas is scarce. This value also prevents the deallocation of private SQL areas associated with open cursors. If the private SQL areas for all concurrently open cursors fills the user's available memory so that there is no space to allocate a private SQL area for a new SQL statement, the statement cannot be parsed and Oracle returns an error indicating that there is not enough memory.

Caching Session Cursors

If an application repeatedly issues parse calls on the same set of SQL statements, the reopening of the session cursors can affect system performance. Session cursors can be stored in a session cursor cache. This feature can be particularly useful for applications designed using Oracle Forms, because switching from one form to another closes all session cursors associated with the first form.

Oracle uses the shared SQL area to determine whether more than three parse requests have been issued on a given statement. If so, Oracle assumes the session cursor associated with the statement should be cached and moves the cursor into the session cursor cache. Subsequent requests to parse that SQL statement by the same session will then find the cursor in the session cursor cache.

To enable caching of session cursors, you must set the initialization parameter SESSION_CACHED_CURSORS. The value of this parameter is a positive integer specifying the maximum number of session cursors kept in the cache. An LRU (Least Recently Used) algorithm removes entries in the session cursor cache to make room for new entries when needed.

You can also enable the session cursor cache dynamically with the statement ALTER SESSION SET SESSION_CACHED_CURSORS.

To determine whether the session cursor cache is sufficiently large for your instance, you can examine the session statistic "session cursor cache hits" in the V$SESSTAT view. This statistic counts the number of times a parse call found a cursor in the session cursor cache. If this statistic is a relatively low percentage of the total parse call count for the session, you should consider setting SESSION_CACHED_CURSORS to a larger value.

Tuning the Data Dictionary Cache

This section describes how to monitor data dictionary cache activity and reduce misses.

Monitoring Data Dictionary Cache Activity

Determine whether misses on the data dictionary cache are affecting the performance of Oracle. You can examine cache activity by querying the V$ROWCACHE table as described in the following sections.

Misses on the data dictionary cache are to be expected in some cases. Upon instance startup, the data dictionary cache contains no data, so any SQL statement issued is likely to result in cache misses. As more data is read into the cache, the likelihood of cache misses should decrease. Eventually the database should reach a "steady state" in which the most frequently used dictionary data is in the cache. At this point, very few cache misses should occur. To tune the cache, examine its activity only after your application has been running.

The V$ROWCACHE View Statistics reflecting data dictionary activity are kept in the dynamic performance table V$ROWCACHE. By default, this table is available only to the user SYS and to users granted SELECT ANY TABLE system privilege, such as SYSTEM.

Each row in this table contains statistics for a single type of the data dictionary item. These statistics reflect all data dictionary activity since the most recent instance startup. These columns in the V$ROWCACHE table reflect the use and effectiveness of the data dictionary cache:

PARAMETER	Identifies a particular data dictionary item. For each row, the value in this column is the item prefixed by dc_. For example, in the row that contains statistics for file descriptions, this column has the value dc_files.
GETS	Shows the total number of requests for information on the corresponding item. For example, in the row that contains statistics for file descriptions, this column has the total number of requests for file descriptions data.

GETMISSES	Shows the number of data requests resulting in cache misses.

Querying the V$ROWCACHE Table Use the following query to monitor the statistics in the V$ROWCACHE table over a period of time while your application is running:

```
SELECT SUM(GETS)  "DATA DICTIONARY GETS",
   SUM(GETMISSES) "DATA DICTIONARY CACHE GET MISSES"
   FROM V$ROWCACHE;
```

The output of this query might look like this:

```
DATA DICTIONARY GETS  DATA DICTIONARY CACHE GET MISSES
--------------------  --------------------------------
             1439044                              3120
```

Interpreting the V$ROWCACHE Table Examining the data returned by the sample query leads to these observations:

- The sum of the GETS column indicates that there were a total of 1,439,044 requests for dictionary data.
- The sum of the GETMISSES column indicates that 3120 of the requests for dictionary data resulted in cache misses.
- The ratio of the sums of GETMISSES to GETS is about 0.2%.

Reducing Data Dictionary Cache Misses

Examine cache activity by monitoring the sums of the GETS and GETMISSES columns. For frequently accessed dictionary caches, the ratio of total GETMISSES to total GETS should be less than 10% or 15%. If the ratio continues to increase above this threshold while your application is running, you should consider increasing the amount of memory available to the data dictionary cache. To increase the memory available to the cache, increase the value of the initialization parameter SHARED_POOL_SIZE. The maximum value for this parameter depends on your operating system.

Tuning the Large Pool and Shared Pool for the Multi-threaded Server Architecture

Oracle recommends allocating memory for MTS session information from the large pool. To do this, specify a value for the parameter LARGE_POOL_SIZE. Otherwise, Oracle allocates memory for MTS from the shared pool. In either case, when using MTS you may need to increase the large pool or shared pool size to accommodate session information.

Note: The Oracle default is dedicated server processing.

When users connect through MTS, Oracle allocates space in the SGA to store information about connections among user processes, dispatchers, and servers. It also stores session information about each user's private SQL area.

The amount of space required in the large or shared pools for each MTS user connection depends on the application.

See Also: Please refer to Chapter 23, "Tuning the Multi-Threaded Server Architecture", *Oracle8i Concepts*, the *Oracle8i Administrator's Guide*, the *Oracle8i SQL Reference*, and the *Net8 Administrator's Guide*.

Reducing Memory Use With Three-Tier Connections

If you have a high number of connected users, you can reduce memory use to an acceptable level by implementing "three-tier connections". This by-product of using a TP monitor is feasible only with pure transactional models, because locks and uncommitted DMLs cannot be held between calls. MTS is much less restrictive of the application design than a TP monitor. It dramatically reduces operating system process count and context switches by enabling users to share a pool of servers. MTS also substantially reduces overall memory usage even though more SGA is used in MTS mode.

Note: On NT, shared servers are implemented as "threads" instead of processes.

The V$SESSTAT View

Oracle collects statistics on total memory used by a session and stores them in the dynamic performance view V$SESSTAT. By default, this view is available only to the user SYS and to users granted SELECT ANY TABLE system privilege, such as SYSTEM. These statistics are useful for measuring session memory use:

session UGA memory	The value of this statistic is the amount of memory in bytes allocated to the session.
session UGA memory max	The value of this statistic is the maximum amount of memory in bytes ever allocated to the session.

To find the value, query V$STATNAME as described in "Technique 3" on page 19-8.

Querying the V$SESSTAT View

You can use this query to decide how much larger to make the shared pool if you are using a Multi-threaded Server. Issue these queries while your application is running:

```
SELECT SUM(VALUE) || ' BYTES' "TOTAL MEMORY FOR ALL SESSIONS"
    FROM V$SESSTAT, V$STATNAME
   WHERE NAME = 'SESSION UGA MEMORY'
      AND V$SESSTAT.STATISTIC# = V$STATNAME.STATISTIC#;
SELECT SUM(VALUE) || ' BYTES' "TOTAL MAX MEM FOR ALL SESSIONS"
   FROM V$SESSTAT, V$STATNAME
   WHERE NAME = 'SESSION UGA MEMORY MAX'
      AND V$SESSTAT.STATISTIC# = V$STATNAME.STATISTIC#;
```

These queries also select from the dynamic performance table V$STATNAME to obtain internal identifiers for *session memory* and *max session memory*. The results of these queries might look like this:

```
TOTAL MEMORY FOR ALL SESSIONS
-----------------------------
157125 BYTES

TOTAL MAX MEM FOR ALL SESSIONS
------------------------------
417381 BYTES
```

Interpreting the V$SESSTAT View

The result of the first query indicates that the memory currently allocated to all sessions is 157,125 bytes. This value is the total memory whose location depends on how the sessions are connected to Oracle. If the sessions are connected to dedicated server processes, this memory is part of the memories of the user processes. If the sessions are connected to shared server processes, this memory is part of the shared pool.

The result of the second query indicates the sum of the maximum sizes of the memories for all sessions is 417,381 bytes. The second result is greater than the first, because some sessions have deallocated memory since allocating their maximum amounts.

You can use the result of either of these queries to determine how much larger to make the shared pool if you use a Multi-threaded Server. The first value is likely to be a better estimate than the second unless nearly all sessions are likely to reach their maximum allocations at the same time.

Tuning Reserved Space from the Shared Pool

On busy systems the database may have difficulty finding a contiguous piece of memory to satisfy a large request for memory. This search may disrupt the behavior of the shared pool, leading to fragmentation and thus affecting performance.

The DBA can reserve memory within the shared pool to satisfy large allocations during operations such as PL/SQL compilation and trigger compilation. Smaller objects will not fragment the reserved list, helping to ensure that the reserved list will have large contiguous chunks of memory. Once the memory allocated from the reserved list is freed, it returns to the reserved list.

Reserved List Tuning Parameters

The size of the reserved list, as well as the minimum size of the objects that can be allocated from the reserved list are controlled by the following initialization parameter:

SHARED_POOL_RESERVED_SIZE	Controls the amount of SHARED_POOL_SIZE reserved for large allocations. The fixed view V$SHARED_POOL_RESERVED helps you tune these parameters. Begin this tuning only after performing all other shared pool tuning.

Controlling Space Reclamation of the Shared Pool

The ABORTED_REQUEST_THRESHOLD procedure, in the package DBMS_SHARED_POOL, lets you limit the size of allocations allowed to flush the shared pool if the free lists cannot satisfy the request size. The database incrementally flushes unused objects from the shared pool until there is sufficient memory to satisfy the allocation request. In most cases, this frees enough memory for the allocation to complete successfully. If the database flushes all objects currently not in use on the system without finding a large enough piece of contiguous memory, an error occurs. Flushing all objects, however, affects other users on the system as well as system performance. The ABORTED_REQUEST_THRESHOLD procedure allows you to localize the error to the process that could not allocate memory.

Initial Parameter Values

Set the initial value of SHARED_POOL_RESERVED_SIZE to 10% of the SHARED_POOL_SIZE. For most systems, this value is sufficient if you have already done some tuning of the shared pool. If you increase this value, then the database allows fewer allocations from the reserved list and requests more memory from the shared pool list.

Ideally, set SHARED_POOL_RESERVED_SIZE large enough to satisfy any request for memory on the reserved list without flushing objects from the shared pool. The amount of operating system memory, however, may constrain the size of the SGA. Making the SHARED_POOL_RESERVED_SIZE large enough to satisfy any request for memory is, therefore, not a feasible goal.

Statistics from the V$SHARED_POOL_RESERVED view can help you tune these parameters. On a system with ample free memory to increase the size of the SGA, the goal is to have REQUEST_MISSES = 0. If the system is constrained for operating system memory, the goal is to not have REQUEST_FAILURES or at least prevent this value from increasing.

If you cannot achieve this, 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: *Oracle8i Reference* for details on setting the LARGE_POOL_SIZE parameter.

SHARED_POOL_ RESERVED_SIZE Too Small

The reserved pool is too small when the value for REQUEST_FAILURES is more than zero and increasing. To resolve this, you can increase the value for the SHARED_POOL_RESERVED_SIZE and SHARED_POOL_SIZE accordingly. The settings you select for these depend on your system's SGA size constraints.

This option increases the amount of memory available on the reserved list without having an effect on users who do not allocate memory from the reserved list. As a second option, reduce the number of allocations allowed to use memory from the reserved list; doing so, however, increases the normal shared pool, which may have an effect on other users on the system.

SHARED_POOL_ RESERVED_SIZE Too Large

Too much memory may have been allocated to the reserved list if:

- REQUEST_MISS = 0 or not increasing
- FREE_MEMORY = > 50% of SHARED_POOL_RESERVED_SIZE minimum

If either of these is true, decrease the value for SHARED_POOL_RESERVED_SIZE.

SHARED_POOL_SIZE Too Small

The V$SHARED_POOL_RESERVED fixed table can also indicate when the value for SHARED_POOL_SIZE is too small. This may be the case if:

- REQUEST_FAILURES > 0 and increasing

Then, if you have enabled the reserved list, decrease the value for SHARED_POOL_RESERVED_SIZE. If you have not enabled the reserved list, you could increase SHARED_POOL_SIZE.

Tuning the Buffer Cache

You can use or bypass the Oracle buffer cache for particular operations. Oracle bypasses the buffer cache for sorting and parallel reads. For operations that use the buffer cache, this section explains:

- Evaluating Buffer Cache Activity by Means of the Cache Hit Ratio
- Increasing the Cache Hit Ratio by Reducing Buffer Cache Misses
- Removing Unnecessary Buffers when Cache Hit Ratio Is High

After tuning private SQL and PL/SQL areas and the shared pool, you can devote the remaining available memory to the buffer cache. It may be necessary to repeat the steps of memory allocation after the initial pass through the process. Subsequent passes allow you to make adjustments in earlier steps based on changes in later steps. For example, if you increase the size of the buffer cache, you may need to allocate more memory to Oracle to avoid paging and swapping.

Evaluating Buffer Cache Activity by Means of the Cache Hit Ratio

Physical I/O takes a significant amount of time, typically in excess of 15 milliseconds. Physical I/O also increases the CPU resources required, owing to the path length in device drivers and operating system event schedulers. Your goal is to reduce this overhead as far as possible by making it more likely that the required block will be in memory. The extent to which you achieve this is measured using the cache hit ratio. Within Oracle this term applies specifically to the database buffer cache.

Calculating the Cache Hit Ratio

Oracle collects statistics that reflect data access and stores them in the dynamic performance view V$SYSSTAT. By default, this table is available only to the user SYS and to users, such as SYSTEM, who have the SELECT ANY TABLE system privilege. Information in the V$SYSSTAT view can also be obtained through the Simple Network Management Protocol (SNMP).

These statistics are useful for tuning the buffer cache:

DB block gets, consistent gets	The sum of the values of these statistics is the total number of requests for data. This value includes requests satisfied by access to buffers in memory.
physical reads	This statistic is the total number of requests for data resulting in access to datafiles on disk.

Monitor these statistics as follows over a period of time while your application is running:

```
SELECT NAME, VALUE
   FROM V$SYSSTAT
   WHERE NAME IN ('DB BLOCK GETS', 'CONSISTENT GETS',
     'PHYSICAL READS');
```

The output of this query might look like this:

```
NAME                                                       VALUE
------------------------------------------------------ ----------
DB BLOCK GETS                                              85792
CONSISTENT GETS                                           278888
PHYSICAL READS                                             23182
```

Calculate the hit ratio for the buffer cache with this formula:

$$\textit{Hit Ratio} = 1 - (\textit{physical reads} / (\textit{db block gets} + \textit{consistent gets}))$$

Based on the statistics obtained by the example query, the buffer cache hit ratio is 94%.

Buffer Pinning Statistics

These statistics are useful in evaluating buffer pinning:

Buffer pinned	This statistic measures the number of times a buffer was already pinned by a client when a client checks to determine if the buffer it wants is already pinned.
Buffer not pinned	This statistic measures the number of times the buffer was not pinned by the client when a client checks to determine if the buffer it wants is already pinned.

These statistics are not incremented when a client performs such a check before releasing it since the client does not intend to use the buffer in this case.

These statistics provide a measure of how often a long consistent read pin on a buffer is beneficial. If the client is able to reuse the pinned buffer many times, it indicates that it is useful to have the buffer pinned.

Evaluating the Cache Hit Ratio

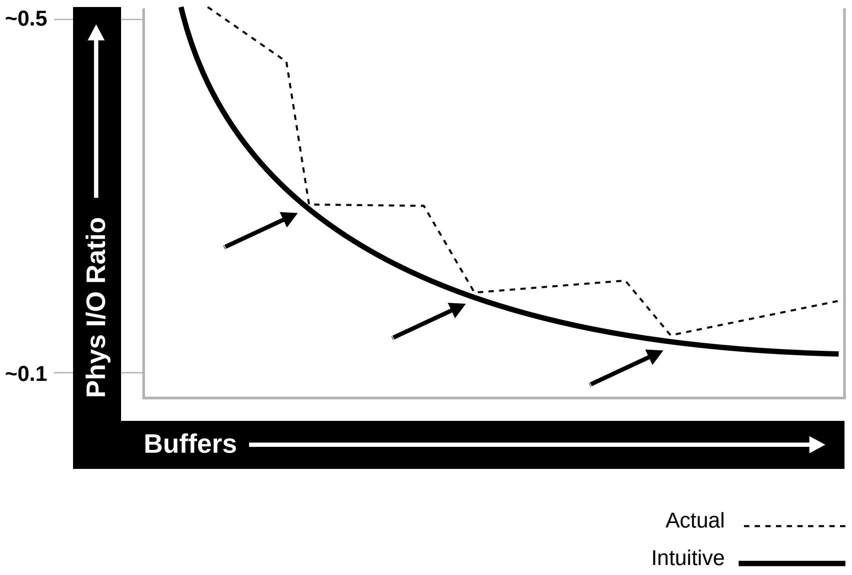
When looking at the cache hit ratio, remember that blocks encountered during a "long" full table scan are not put at the head of the LRU list; therefore repeated scanning does not cause the blocks to be cached.

Repeated scanning of the same large table is rarely the most efficient approach. It may be better to perform all of the processing in a single pass, even if this means that the overnight batch suite can no longer be implemented as a SQL*Plus script that contains no PL/SQL. The solution therefore lies at the design or implementation level.

Production sites running with thousands or tens of thousands of buffers rarely use memory effectively. In any large database running OLTP applications, in any given unit of time, most rows are accessed either one or zero times. On this basis there is little point in keeping the row, or the block that contains it, in memory for very long following its use.

Finally, the relationship between the cache hit ratio and the number of buffers is far from a smooth distribution. When tuning the buffer pool, avoid the use of additional buffers that contribute little or nothing to the cache hit ratio. As illustrated in Figure 19–2, only narrow bands of values of DB_BLOCK_BUFFERS are worth considering.

Figure 19–2 Buffer Pool Cache Hit Ratio



Note: A common mistake is to continue increasing the value of DB_BLOCK_BUFFERS. Such increases have no effect if you are doing full table scans or other operations that do not use the buffer pool.

As a rule of thumb, increase DB_BLOCK_BUFFERS when:

- The cache hit ratio is less than 0.9
- There is no evidence of undue page faulting
- The previous increase of DB_BLOCK_BUFFERS was effective

Determining Which Buffers Are in the Pool

The CATPARR.SQL script creates the view V$BH, which shows the file number and block number of blocks that currently reside within the SGA. Although CATPARR.SQL is primarily intended for use in parallel server environments, you can run it as SYS even if you're operating a single instance environment.

Perform a query similar to the following:

```
SELECT file#, COUNT(block#), COUNT (DISTINCT file# || block#)
FROM V$BH
GROUP BY file#;
```

Increasing the Cache Hit Ratio by Reducing Buffer Cache Misses

If your hit ratio is low, or less than 60% or 70%, then you may want to increase the number of buffers in the cache to improve performance. To make the buffer cache larger, increase the value of the initialization parameter DB_BLOCK_BUFFERS.

Oracle can collect statistics that estimate the performance gain that would result from increasing the size of your buffer cache. With these statistics, you can estimate how many buffers to add to your cache.

Removing Unnecessary Buffers when Cache Hit Ratio Is High

If your hit ratio is high, your cache is probably large enough to hold your most frequently accessed data. In this case, you may be able to reduce the cache size and still maintain good performance. To make the buffer cache smaller, reduce the value of the initialization parameter DB_BLOCK_BUFFERS. The minimum value for this parameter is 4. You can use any leftover memory for other Oracle memory structures.

Oracle can collect statistics to predict buffer cache performance based on a smaller cache size. Examining these statistics can help you determine how small you can afford to make your buffer cache without adversely affecting performance.

Accommodating LOBs in the Buffer Cache

Both temporary and permanent LOBs can use the buffer cache.

Temporary LOBs

Temporary LOBS created when you have set the CACHE parameter to TRUE move through the buffer cache. Temporary LOBS created with the CACHE parameter set to FALSE are read directly from and written directly to disk.

You can use durations for automatic cleanup to save time and effort. Also, it is more efficient for the database to end a duration and free all temporary LOBs associated with a duration than it is to free each one explicitly.

Temporary LOBs create deep copies of themselves on assignments. For example:

```
LOCATOR1 BLOB;
LOCATOR2 BLOB;
DBMS_LOB.CREATETEMPORARY (LOCATOR1,TRUE,DBMS_LOB.SESSION);
LOCATOR2 := LOCATOR;
```

The above code causes a copy of the temporary LOB pointed to by LOCATOR1 to be created. You may also want to consider using pass by reference semantics in PL/SQL.

Or, in OCI, you may declare pointers to locators as in the following example:

```
OCILOBDESCRIPTOR *LOC1;
OCILOBDESCRIPTOR *LOC2;
OCILOBCREATETEMPORARY (LOC1,TRUE,OCIDURATIONSESSION);
LOC2 = LOC1;
```

Avoid using OCILobAssign() commands as these also cause deep copies of temporary LOBs. In other words, a new copy of the temporary LOB is created.

Pointer assignment does not cause deep copies, it just causes pointers to point to the same thing.

Tuning Multiple Buffer Pools

This section covers:

- Overview of the Multiple Buffer Pool Feature
- When to Use Multiple Buffer Pools

Overview of the Multiple Buffer Pool Feature

Schema objects are referenced with varying usage patterns; therefore, their cache behavior may be quite different. Multiple buffer pools enable you to address these differences. You can use a KEEP buffer pool to maintain objects in the buffer cache and a RECYCLE buffer pool to prevent objects from consuming unnecessary space in the cache. When an object is allocated to a cache, all blocks from that object are placed in that cache. Oracle maintains a DEFAULT buffer pool for objects that have not been assigned to one of the buffer pools.

Each buffer pool in Oracle comprises a number of working sets. A different number of sets can be allocated for each buffer pool. All sets use the same LRU (Least Recently Used) replacement policy. A strict LRU aging policy provides good hit rates in most cases, but you can sometimes improve hit rates by providing some hints.

The main problem with the LRU list occurs when a very large segment is accessed frequently in a random fashion. Here, "very large" means large compared to the size of the cache. Any single segment that accounts for a substantial portion (more than 10%) of nonsequential physical reads is probably one of these segments. Random reads to such a large segment can cause buffers that contain data for other segments to be aged out of the cache. The large segment ends up consuming a large percentage of the cache, but does not benefit from the cache.

Very frequently accessed segments are not affected by large segment reads, because their buffers are warmed frequently enough that they do not age out of the cache. The main trouble occurs with "warm" segments that are not accessed frequently enough to survive the buffer flushing caused by the large segment reads.

You have two options for solving this problem. One is to move the large segment into a separate RECYCLE cache so that it does not disturb the other segments. The RECYCLE cache should be smaller than the DEFAULT buffer pool and should reuse buffers more quickly than the DEFAULT buffer pool.

The other approach is to move the small warm segments into a separate KEEP cache that is not used at all for large segments. The KEEP cache can be sized to minimize misses in the cache. You can make the response times for specific queries more predictable by putting the segments accessed by the queries in the KEEP cache to ensure that they are never aged out.

When to Use Multiple Buffer Pools

When you examine system I/O performance, you should analyze the schema and determine whether multiple buffer pools would be advantageous. Consider a KEEP cache if there are small, frequently accessed tables that require quick response time. Very large tables with random I/O are good candidates for a RECYCLE cache.

Use the following steps to determine the percentage of the cache used by an individual object at a given point in time:

1. Find the Oracle internal object number of the segment by entering:

   ```
   SELECT DATA_OBJECT_ID, OBJECT_TYPE FROM USER_OBJECTS
   WHERE OBJECT_NAME = '<SEGMENT_NAME>';
   ```

   Because two objects can have the same name (if they are different types of objects), you can use the OBJECT_TYPE column to identify the object of interest. If the object is owned by another user, then use the view DBA_OBJECTS or ALL_OBJECTS instead of USER_OBJECTS.

2. Find the number of buffers in the buffer cache for SEGMENT_NAME:

   ```
   SELECT COUNT(*) BUFFERS FROM V$BH WHERE OBJD = <DATA_OBJECT_ID>;
   ```

   where DATA_OBJECT_ID is from Step 1.

3. Find the total number of buffers in the instance:

   ```
   SELECT VALUE "TOTAL BUFFERS" FROM  V$PARAMETER
   WHERE NAME = 'DB_BLOCK_BUFFERS';
   ```

4. Calculate the ratio of buffers to total buffers, to obtain the percentage of the cache currently used by SEGMENT_NAME.

$$\text{\% cache used by } \textit{segment\_name} = \frac{\textit{buffers} \quad \text{(Step 2)}}{\textit{total buffers} \quad \text{(Step 3)}}$$

Note: This technique works only for a single segment. You must run the query for each partition for a partitioned object.

If the number of local block gets equals the number of physical reads for statements involving such objects, consider using a RECYCLE cache because of the limited usefulness of the buffer cache for the objects.

Tuning the Buffer Cache Using Multiple Buffer Pools

When you partition your buffer cache into multiple buffer pools, each buffer pool can be used for blocks from objects that are accessed in different ways. If the blocks of a particular object are likely to be reused, then you should pin that object in the buffer cache so the next use of the block does not require disk I/O. Conversely, if a block probably will not be reused within a reasonable period of time, discard it to make room for more frequently used blocks.

By properly allocating objects to appropriate buffer pools, you can:

- Reduce or eliminate I/Os.
- Isolate an object in the cache.
- Restrict or limit an object to a part of the cache.

Enabling Multiple Buffer Pools

You can create multiple buffer pools for each database instance. The same set of buffer pools need not be defined for each instance of the database. Among instances, the buffer pools may be different sizes or not defined at all. Tune each instance separately.

Defining New Buffer Pools

You can define each buffer pool using the BUFFER_POOL_*name* initialization parameter. You can specify two attributes for each buffer pool: the number of buffers in the buffer pool and the number of LRU latches allocated to the buffer pool.

The initialization parameters used to define buffer pools are:

BUFFER_POOL_KEEP	Defines the KEEP buffer pool.
BUFFER_POOL_RECYCLE	Defines the RECYCLE buffer pool.
DB_BLOCK_BUFFERS	Defines the number of buffers for the database instance. Each individual buffer pool is created from this total amount with the remainder allocated to the DEFAULT buffer pool.
DB_BLOCK_LRU_LATCHES	Defines the number of LRU latches for the entire database instance. Each buffer pool defined takes from this total in a fashion similar to DB_BLOCK_BUFFERS.

For example:

```
BUFFER_POOL_KEEP=(buffers:400, lru_latches:3")
BUFFER_POOL_RECYCLE=(buffers:50, lru_latches:1")
```

The size of each buffer pool is subtracted from the total number of buffers defined for the entire buffer cache (that is, the value of the `DB_BLOCK_BUFFERS parameter`). The aggregate number of buffers in all buffer pools cannot, therefore, exceed this value. Likewise, the number of LRU latches allocated to each buffer pool is taken from the total number allocated to the instance by the DB_BLOCK_LRU_LATCHES parameter. If either constraint is violated, Oracle displays an error and the database is not mounted.

The minimum number of buffers you must allocate to each buffer pool is 50 times the number of LRU latches. For example, a buffer pool with 3 LRU latches must have at least 150 buffers.

Oracle automatically defines three buffer pools: KEEP, RECYCLE, and DEFAULT. The DEFAULT buffer pool always exists. You do not explicitly define the size of the DEFAULT buffer pool or the number of working sets assigned to the DEFAULT buffer pool. Rather, each value is inferred from the total number allocated minus the number allocated to every other buffer pool. There is no requirement that any one buffer pool be defined for another buffer pool to be used.

Using Multiple Buffer Pools

This section describes how to establish a DEFAULT buffer pool for an object. All blocks for the object go in the specified buffer pool.

The BUFFER_POOL clause is used to define the DEFAULT buffer pool for an object. This clause is valid for CREATE and ALTER table, cluster, and index DDL statements. The buffer pool name is case insensitive. The blocks from an object without an explicitly set buffer pool go into the DEFAULT buffer pool.

If a buffer pool is defined for a partitioned table or index, each partition of the object inherits the buffer pool from the table or index definition unless you override it with a specific buffer pool.

When the DEFAULT buffer pool of an object is changed using the `ALTER` statement, all buffers currently containing blocks of the altered segment remain in the buffer pool they were in before the ALTER statement. Newly loaded blocks and any blocks that have aged out and are reloaded go into the new buffer pool.

The syntax of the BUFFER_POOL clause is: `BUFFER_POOL` {`KEEP` | `RECYCLE` | `DEFAULT`}

For example:

```
BUFFER_POOL KEEP
```

Or:

```
BUFFER_POOL RECYCLE
```

The following DDL statements accept the buffer pool clause:

- `CREATE TABLE` *table name...* STORAGE *(buffer_pool_clause)*

  A buffer pool is not permitted for a clustered table. The buffer pool for a clustered table is specified at the cluster level.

  For an index-organized table, a buffer pool can be defined on both the index and the overflow segment.

  For a partitioned table, a buffer pool can be defined on each partition. The buffer pool is specified as a part of the storage clause for each partition.

For example:

```
CREATE TABLE TABLE_NAME (COL_1 NUMBER, COL_2 NUMBER)
PARTITION BY RANGE (COL_1)
(PARTITION ONE VALUES LESS THAN (10)
STORAGE (INITIAL 10K BUFFER_POOL RECYCLE),
PARTITION TWO VALUES LESS THAN (20) STORAGE (BUFFER_POOL KEEP));
```

- `CREATE INDEX` *index name*... STORAGE *(buffer_pool_clause)*

  For a global or local partitioned index, a buffer pool can be defined on each partition.

- `CREATE CLUSTER` *cluster_name*...STORAGE (*buffer_pool_clause)*
- `ALTER TABLE` *table_name*... STORAGE (*buffer_pool_clause)*

  A buffer pool can be defined during simple ALTER TABLE, MODIFY PARTITION, MOVE PARTITION, ADD PARTITION, and SPLIT PARTITION commands for both new partitions.

- `ALTER INDEX` *index_name*... STORAGE (*buffer_pool_clause)*

  A buffer pool can be defined during simple ALTER INDEX, REBUILD, MODIFY PARTITION, SPLIT PARTITION commands for both new partitions, and rebuild partitions.

- `ALTER CLUSTER` *cluster_name*... STORAGE (*buffer_pool_clause)*

Dictionary Views Showing Default Buffer Pools

The following dictionary views have a BUFFER POOL column indicating the DEFAULT buffer pool for the given object.

USER_CLUSTERS	ALL_CLUSTERS	DBA_CLUSTERS
USER_INDEXES	ALL_INDEXES	DBA_INDEXES
USER_SEGMENTS	DBA_SEGMENTS	
USER_TABLES	USER_OBJECT_TABLES	USER_ALL_TABLES
ALL_TABLES	ALL_OBJECT_TABLES	ALL_ALL_TABLES
DBA_TABLES	DBA_OBJECT_TABLES	DBA_ALL_TABLES
USER_PART_TABLES	ALL_PART_TABLES	DBA_PART_TABLES
USER_PART_INDEXES	ALL_PART_INDEXES	DBA_PART_INDEXES
USER_TAB_PARTITIONS	ALL_TAB_PARTITIONS	DBA_TAB_PARTITIONS
USER_IND_PARTITIONS	ALL_IND_PARTITIONS	DBA_IND_PARTITIONS

The views V$BUFFER_POOL_STATISTICS and GV$BUFFER_POOL_STATISTICS describe the buffer pools allocated on the local instance and entire database, respectively. To create these views you must run the CATPERF.SQL file.

Sizing Each Buffer Pool

This section explains how to size the KEEP and RECYCLE buffer pools.

KEEP Buffer Pool

The goal of the KEEP buffer pool is to retain objects in memory, thus avoiding I/O operations. The size of the KEEP buffer pool therefore depends on the objects that you wish to keep in the buffer cache. You can compute an approximate size for the KEEP buffer pool by adding together the sizes of all objects dedicated to this pool. Use the ANALYZE command to obtain the size of each object. Although the ESTIMATE option provides a rough measurement of sizes, the COMPUTE STATISTICS option is preferable because it provides the most accurate value possible.

The buffer pool hit ratio can be determined using the formula:

$$\textit{hit ratio} = 1 - \frac{\textit{physical reads}}{(\textit{block gets} + \textit{consistent gets})}$$

Where the values of physical reads, block gets, and consistent gets can be obtained for the KEEP buffer pool from the following query:

```
SELECT PHYSICAL_READS, BLOCK_GETS, CONSISTENT_GETS
FROM V$BUFFER_POOL_STATISTICS WHERE NAME = 'KEEP';
```

The KEEP buffer pool will have a 100% hit ratio only after the buffers have been loaded into the buffer pool. Therefore, do not compute the hit ratio until after the system runs for a while and achieves steady-state performance. Calculate the hit ratio by taking two snapshots of system performance at different times using the above query. Subtract the newest values from the older values for physical reads, block gets, and consistent gets and use these values to compute the hit ratio.

A 100% buffer pool hit ratio may not be optimal. Often you can decrease the size of your KEEP buffer pool and still maintain a sufficiently high hit ratio. Allocate blocks removed from use for the KEEP buffer pool to other buffer pools.

Note: If an object grows in size, then it may no longer fit in the KEEP buffer pool. You will begin to lose blocks out of the cache.

Each object kept in memory results in a trade-off: it is beneficial to keep frequently accessed blocks in the cache, but retaining infrequently used blocks results in less space for other, more active blocks.

RECYCLE Buffer Pool

The goal of the RECYCLE buffer pool is to eliminate blocks from memory as soon as they are no longer needed. If an application accesses the blocks of a very large object in a random fashion, there is little chance of reusing a block stored in the buffer pool before it is aged out. This is true regardless of the size of the buffer pool (given the constraint of the amount of available physical memory). Because of this, the object's blocks should not be cached; those cache buffers can be allocated to other objects.

Be careful, however, not to discard blocks from memory too quickly. If the buffer pool is too small, blocks may age out of the cache before the transaction or SQL statement has completed execution. For example, an application may select a value from a table, use the value to process some data, and then update the record. If the block is removed from the cache after the select statement, it must be read from disk again to perform the update. The block should be retained for the duration of the user transaction.

By executing statements with a SQL statement tuning tool such as Oracle Trace or with the SQL trace facility enabled and running TKPROF on the trace files, you can get a listing of the total number of data blocks physically read from disk. (This number appears in the "disk" column in the TKPROF output.) The number of disk reads for a particular SQL statement should not exceed the number of disk reads of the same SQL statement with all objects allocated from the DEFAULT buffer pool.

Two other statistics can tell you whether the RECYCLE buffer pool is too small. If the "free buffer waits" statistic ever becomes excessive, the pool is probably too small. Likewise, the number of "log file sync" wait events will increase. One way to size the RECYCLE buffer pool is to run the system with the RECYCLE buffer pool disabled. At steady state, the number of buffers in the DEFAULT buffer pool being consumed by segments that would normally go in the RECYCLE buffer pool can be divided by four. Use the result as a value for sizing the RECYCLE cache.

Identifying Segments to Put into the KEEP and RECYCLE Buffer Pools

A good candidate for a segment to put into the RECYCLE buffer pool is a segment that is at least twice the size of the DEFAULT buffer pool and has incurred at least a few percent of the total I/Os in the system.

A good candidate for a segment to put into the KEEP pool is a segment that is smaller than 10% of the size of the DEFAULT buffer pool and has incurred at least 1% of the total I/Os in the system.

The trouble with these rules is that it can sometimes be difficult to determine the number of I/Os per segment if a tablespace has more than one segment. One way to solve this problem is to sample the I/Os that occur over a period of time by selecting from V$SESSION_WAIT to determine a statistical distribution of I/Os per segment.

Another option is to examine the positions of the blocks of a segment in the buffer cache. In particular, the ratio of the count of blocks for a segment in the hot half of the cache to the count in the cold half for the same segment can give a good indication of which segments are hot and which are cold. If the ratio for a segment is close to 1, then buffers for that segment are not frequently used and the segment may be a good candidate for the RECYCLE cache. If the ratio is high, perhaps 3.0, then the buffers are frequently used and the segment might be a good candidate for the KEEP cache.

Identifying and Eliminating LRU Latch Contention

LRU latches regulate the least recently used (LRU) buffer lists used by the buffer cache. If there is latch contention then processes are waiting and spinning before obtaining the latch.

You can set the overall number of latches in the database instance using the DB_BLOCK_LRU_LATCHES parameter. When each buffer pool is defined, a number of these LRU latches can be reserved for the buffer pool. The buffers of a buffer pool are divided evenly between the LRU latches of the buffer pool.

To determine whether your system is experiencing latch contention, begin by determining whether there is LRU latch contention for any individual latch.

```
SELECT child#, sleeps / gets ratio
 FROM V$LATCH_CHILDREN
 WHERE name = 'cache buffers lru chain';
```

The miss ratio for each LRU latch should be less than 1%. A ratio above 1% for any particular latch is indicative of LRU latch contention and should be addressed. You can determine the buffer pool to which the latch is associated as follows:

```
SELECT name FROM V$BUFFER_POOL_STATISTICS
 WHERE lo_setid <= child_latch_number
 AND hi_setid >= child_latch_number;
```

where *child_latch_number* is the *child#* from the previous query.

You can alleviate LRU latch contention by increasing the overall number of latches in the system and the number of latches allocated to the buffer pool indicated in the second query.

The maximum number of latches allowed is the lower of:

$$number\_of\_cpus * 2 * 3 \quad \text{or} \quad number\_of\_buffers / 50$$

This limitation exists because no set can have fewer than 50 buffers. If you specify a value larger than the maximum, Oracle automatically resets the number of latches to the largest value allowed by the formula.

For example, if the number of CPUs is 4 and the number of buffers is 200, then a maximum of 4 latches would be allowed (minimum of 4*2*3, 200/50). If the number of CPUs is 4 and the number of buffers is 10000, then the maximum number of latches allowed is 24 (minimum of 4*2*3, 10000/50).

Tuning Sort Areas

If large sorts occur frequently, consider increasing the value of the parameter SORT_AREA_SIZE with either or both of two goals in mind:

- To increase the number of sorts that can be conducted entirely within memory.
- To speed up those sorts that cannot be conducted entirely within memory.

Large sort areas can be used effectively if you combine a large SORT_AREA_SIZE with a minimal SORT_AREA_RETAINED_SIZE. If memory is not released until the user disconnects from the database, large sort work areas could cause problems. The SORT_AREA_RETAINED_SIZE parameter lets you specify the level down to which memory should be released as soon as possible following the sort. Set this parameter to zero if large sort areas are being used in a system with many simultaneous users.

SORT_AREA_RETAINED_SIZE is maintained for each sort operation in a query. Thus if 4 tables are being sorted for a sort merge, Oracle maintains 4 areas of SORT_AREA_RETAINED_SIZE.

See Also: Chapter 26, "Tuning Parallel Execution".

Reallocating Memory

After resizing your Oracle memory structures, re-evaluate the performance of the library cache, the data dictionary cache, and the buffer cache. If you have reduced the memory consumption of any of these structures, you may want to allocate more memory to another. For example, if you have reduced the size of your buffer cache, you may want to use the additional memory by for the library cache.

Tune your operating system again. Resizing Oracle memory structures may have changed Oracle memory requirements. In particular, be sure paging and swapping are not excessive. For example, if the size of the data dictionary cache or the buffer cache has increased, the SGA may be too large to fit into main memory. In this case, the SGA could be paged or swapped.

While reallocating memory, you may determine that the optimum size of Oracle memory structures requires more memory than your operating system can provide.

In this case, you may improve performance even further by adding more memory to your computer.

Reducing Total Memory Usage

If the overriding performance problem is that the server simply does not have enough memory to run the application as currently configured, and the application is logically a single application (that is, it cannot readily be segmented or distributed across multiple servers), then only two possible solutions exist:

- Increase the amount of memory available.
- Decrease the amount of memory used.

The most dramatic reductions in server memory usage always come from reducing the number of database connections, which in turn can resolve issues relating to the number of open network sockets and the number of operating system processes. However, to reduce the number of connections without reducing the number of users, the connections that remain must be shared. This forces the user processes to adhere to a paradigm in which every message request sent to the database describes a complete or "atomic" transaction.

Writing applications to conform to this model is not necessarily either restrictive or difficult, but it is certainly different. Conversion of an existing application, such as an Oracle Forms suite, to conform is not normally possible without a complete rewrite.

The Oracle Multi-threaded Server architecture is an effective solution for reducing the number of server operating system processes. MTS is also quite effective at reducing overall memory requirements. You can also use MTS to reduce the number of network connections when you use MTS with connection pooling and connection multiplexing.

Shared connections are possible in Oracle Forms environments when you use an intermediate server that is also a client. In this configuration, use the DBMS_PIPE mechanism to transmit atomic requests from the user's individual connection on the intermediate server to a shared daemon in the intermediate server. The daemon, in turn, owns a connection to the central server.

20 Tuning I/O

This chapter explains how to avoid I/O bottlenecks that could prevent Oracle from performing at its maximum potential. This chapter covers the following topics:

Understanding I/O Problems

This section introduces I/O performance issues. It covers:

The performance of many software applications is inherently limited by disk input/output (I/O). Often, CPU activity must be suspended while I/O activity completes. Such an application is said to be "I/O bound". Oracle is designed so that performance need not be limited by I/O.

Tuning I/O can enhance performance if a disk containing database files is operating at its capacity. However, tuning I/O cannot help performance in "CPU bound" cases—or cases in which your computer's CPUs are operating at their capacity.

It is important to tune I/O after following the recommendations presented in Chapter 19, "Tuning Memory Allocation". That chapter explains how to allocate memory so as to reduce I/O to a minimum. After reaching this minimum, follow the instructions in this chapter to achieve more efficient I/O performance.

Tuning I/O: Top Down and Bottom Up

When designing a new system, you should analyze I/O needs from the top down, determining what resources you will require in order to achieve the desired performance.

For an existing system, you should approach I/O tuning from the bottom up:

1. Determine the number of disks on the system.
2. Determine the number of disks that are being used by Oracle.
3. Determine the type of I/Os that your system performs.
4. Ascertain whether the I/Os are going to the file system or to raw devices.
5. Determine how to spread objects over multiple disks, using either manual striping or striping software.
6. Calculate the level of performance you can expect.

Analyzing I/O Requirements

This section explains how to determine your system's I/O requirements.

1. Calculate the total throughput your application will require.

   Begin by figuring out the number of reads and writes involved in each transaction, and distinguishing the objects against which each operation is performed.

In an OLTP application, for example, each transaction might involve:

- 1 read from object A
- 1 read from object B
- 1 write to object C

One transaction in this example thus requires 2 reads and 1 write, all to different objects.

2. Define the I/O performance target for this application by specifying the number of tps (transactions per second) the system must support.

   In this example, the designer might specify that 100 tps would constitute an acceptable level of performance. To achieve this, the system must be able to perform 300 I/Os per second:

   - 100 reads from object A
   - 100 reads from object B
   - 100 writes to object C

3. Determine the number of disks needed to achieve this level of performance.

   To do this, ascertain the number of I/Os that each disk can perform per second. This numbers depends on three factors:

   - Speed of your particular disk hardware
   - Whether the I/Os needed are reads or writes
   - Whether you are using the file system or raw devices

   In general, disk speed tends to have the following characteristics:

Table 20–1 Relative Disk Speed

Disk Speed:	File System	Raw Devices
Reads per second	fast	slow
Writes per second	slow	fast

4. Write the relative speed per operation of your disks in a chart like the one shown in Table 20–2:

Table 20–2 Disk I/O Analysis Worksheet

Disk Speed:	File System	Raw Devices
Reads per second		
Writes per second		

5. The disks in the current example have characteristics as shown in Table 20–3:

Table 20–3 Sample Disk I/O Analysis

Disk Speed:	File System	Raw Devices
Reads per second	50	45
Writes per second	20	50

6. Calculate the number of disks you need to achieve your I/O performance target using a chart like the one shown in Table 20–4:

Table 20–4 Disk I/O Requirements Worksheet

	If Stored on File System			If Stored on Raw Devices		
Object	R/W Needed per Sec.	Disk R/W Capabil. per Sec.	Disks Needed	R/W Needed per Sec.	Disk R/W Capabil. per Sec.	Disks Needed
A						
B						
C						
Disks Req'd						

Table 20–5 shows the values from this example:

Table 20–5 Sample Disk I/O Requirements

	If Stored on File System			If Stored on Raw Devices		
Object	**R/W Needed per Sec.**	**Disk R/W Capabil. per Sec.**	**Disks Needed**	**R/W Needed per Sec.**	**Disk R/W Capabil. per Sec.**	**Disks Needed**
A	100 reads	50 reads	2 disks	100 reads	45 reads	2 disks
B	100 reads	50 reads	2 disks	100 reads	45 reads	2 disks
C	100 writes	20 writes	5 disks	100 writes	50 writes	2 disks
Disks Req'd			9 disks			6 disks

Planning File Storage

This section explains how to determine whether your application will run best by:

- Running the application on the disks available
- Storing the data on raw devices
- Storing the data on block devices
- Storing the data directly on the file system

Design Approach

Use the following approach to design file storage:

1. Identify the operations required by your application.
2. Test the performance of your system's disks for the different operations required by your application.
3. Finally, evaluate what kind of disk layout will give you the best performance for the operations that predominate in your application.

These steps are described in detail under the following headings.

Identifying the Required Read/Write Operations

Evaluate your application to determine how often it requires each type of I/O operation. Table 20–6 shows the types of read and write operations performed by each of the background processes, by foreground processes, and by parallel execution servers.

Table 20–6 Read/Write Operations Performed by Oracle Processes

	Process							
Operation	**LGWR**	**DBW*n***	**ARCH**	**SMON**	**PMON**	**CKPT**	**Fore-ground**	**PQ Processes**
Sequential Read			X	X		X	X	X
Sequential Write	X		X			X		
Random Read				X			X	
Random Write		X						

In this discussion, a sample application might involve 50% random reads, 25% sequential reads, and 25% random writes.

Testing the Performance of Your Disks

This section illustrates relative performance of read / write operations by a particular test system. On raw devices, reads and writes are done on the character level; on block devices, these operations are done on the block level. (Many concurrent processes may generate overhead due to head and arm movement of the disk drives.)

Note: Values provided in this example *do not* constitute a rule of thumb. They were generated by an actual UNIX test system using particular disks. *These figures will differ significantly for different platforms and different disks!* To make accurate judgments, *test your own system* using an approach similar to the one demonstrated in this section. Alternatively, contact your system vendor for information on disk performance for the different operations.

Table 20–7 and Figure 20–1 show speed of sequential read in milliseconds per I/O, for each of the three disk layout options on a test system.

Table 20–7 Block Size and Speed of Sequential Read (Sample Data)

	Speed of Sequential Read on:		
Block Size	**Raw Device**	**Block Device**	**UNIX File System**
512 bytes	1.4	0.6	0.4
1KB	1.4	0.6	0.3
2KB	1.5	1.1	0.6
4KB	1.6	1.8	1.0
8KB	2.7	3.0	1.5
16KB	5.1	5.3	3.7
32KB	10.1	10.3	8.1
64KB	20.0	20.3	18.0
128KB	40.4	41.3	36.1
256KB	80.7	80.3	61.3

Doing research like this helps determine the correct stripe size. In this example, it takes at most 5.3 milliseconds to read 16KB. If your data were in chunks of 256KB, you could stripe the data over 16 disks (as described on page 20-22) and maintain this low read time.

By contrast, if all your data were on one disk, read time would be 80 milliseconds. Thus the test results show that on this particular set of disks, things look quite different from what might be expected: it is sometimes beneficial to have a smaller stripe size, depending on the size of the I/O.

Figure 20–1 Block Size and Speed of Sequential Read (Sample Data)

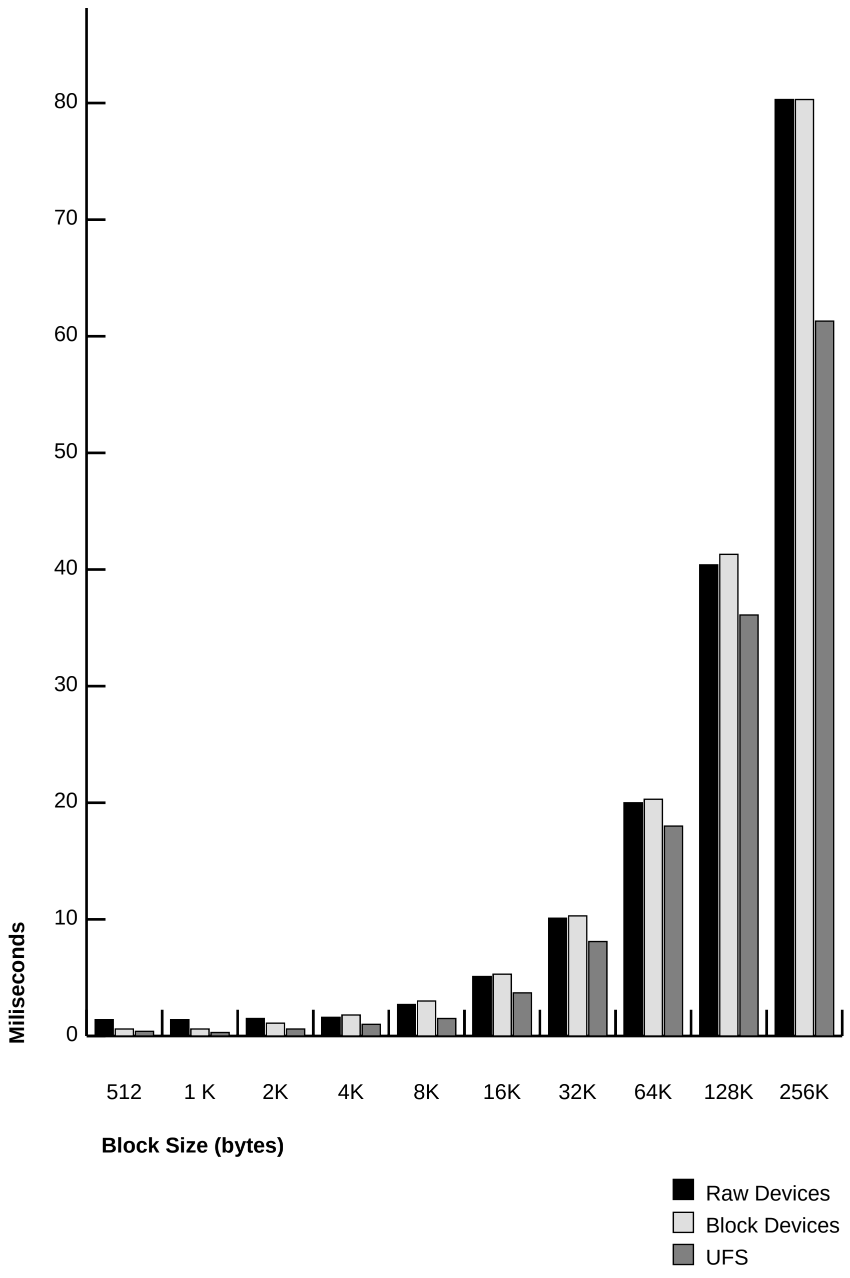


Table 20–8 and Figure 20–2 show speed of sequential write in milliseconds per I/O, for each of the three disk layout options on the test system.

Table 20–8 Block Size and Speed of Sequential Write (Sample Data)

	Speed of Sequential Write on		
Block Size	**Raw Device**	**Block Device**	**UNIX File System**
512 bytes	11.2	11.8	17.9
1KB	11.7	11.9	18.3
2KB	11.6	13.0	19.0
4KB	12.3	13.8	19.8
8KB	13.5	13.8	21.8
16KB	16.0	27.8	35.3
32KB	19.3	55.6	62.2
64KB	31.5	111.1	115.1
128KB	62.5	224.5	221.8
256KB	115.6	446.1	429.0

Figure 20–2 Block Size and Speed of Sequential Write (Sample Data)

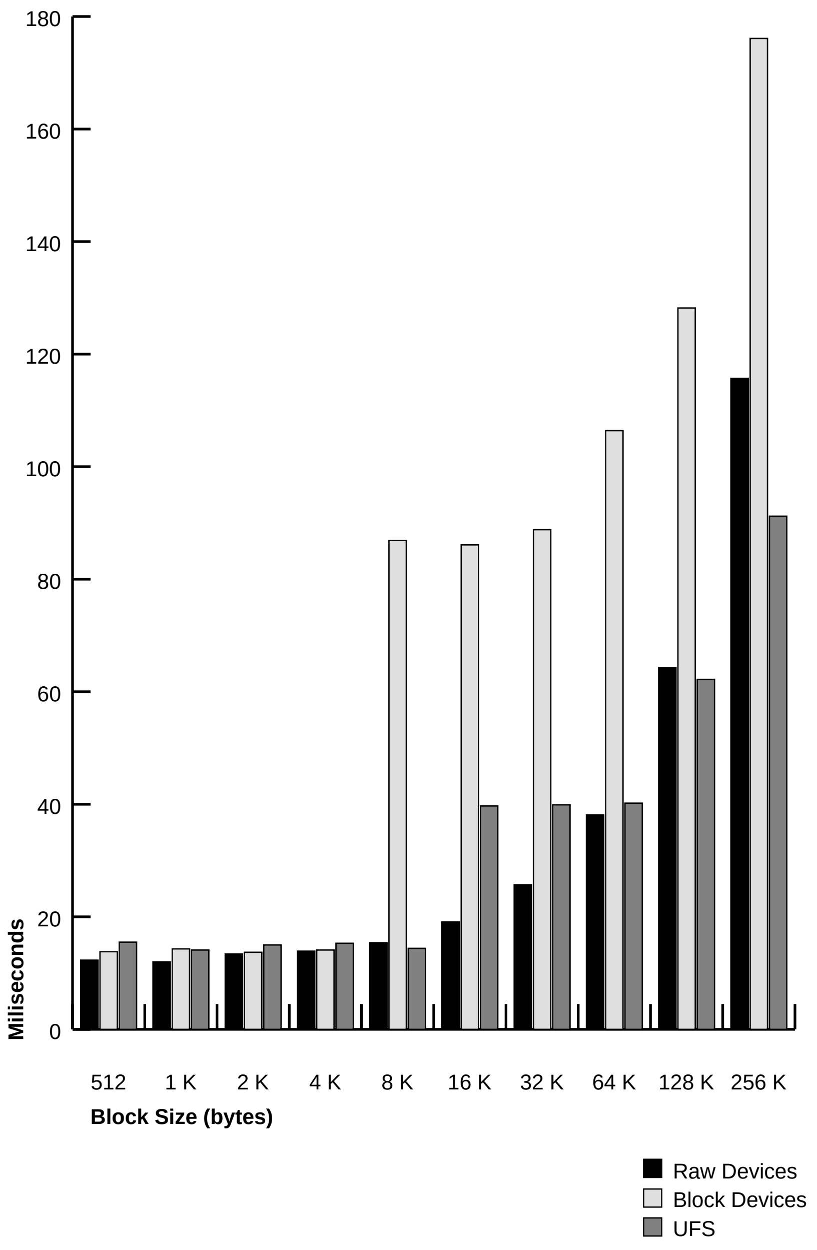


Table 20–9 and Figure 20–3 show speed of random read in milliseconds per I/O, for each of the three disk layout options on the test system.

Table 20–9 Block Size and Speed of Random Read (Sample Data)

	Speed of Random Read on		
Block Size	**Raw Device**	**Block Device**	**UNIX File System**
512 bytes	12.3	13.8	15.5
1KB	12.0	14.3	14.1
2KB	13.4	13.7	15.0
4KB	13.9	14.1	15.3
8KB	15.4	86.9	14.4
16KB	19.1	86.1	39.7
32KB	25.7	88.8	39.9
64KB	38.1	106.4	40.2
128KB	64.3	128.2	62.2
256KB	115.7	176.1	91.2

Figure 20–3 Block Size and Speed of Random Read (Sample Data)

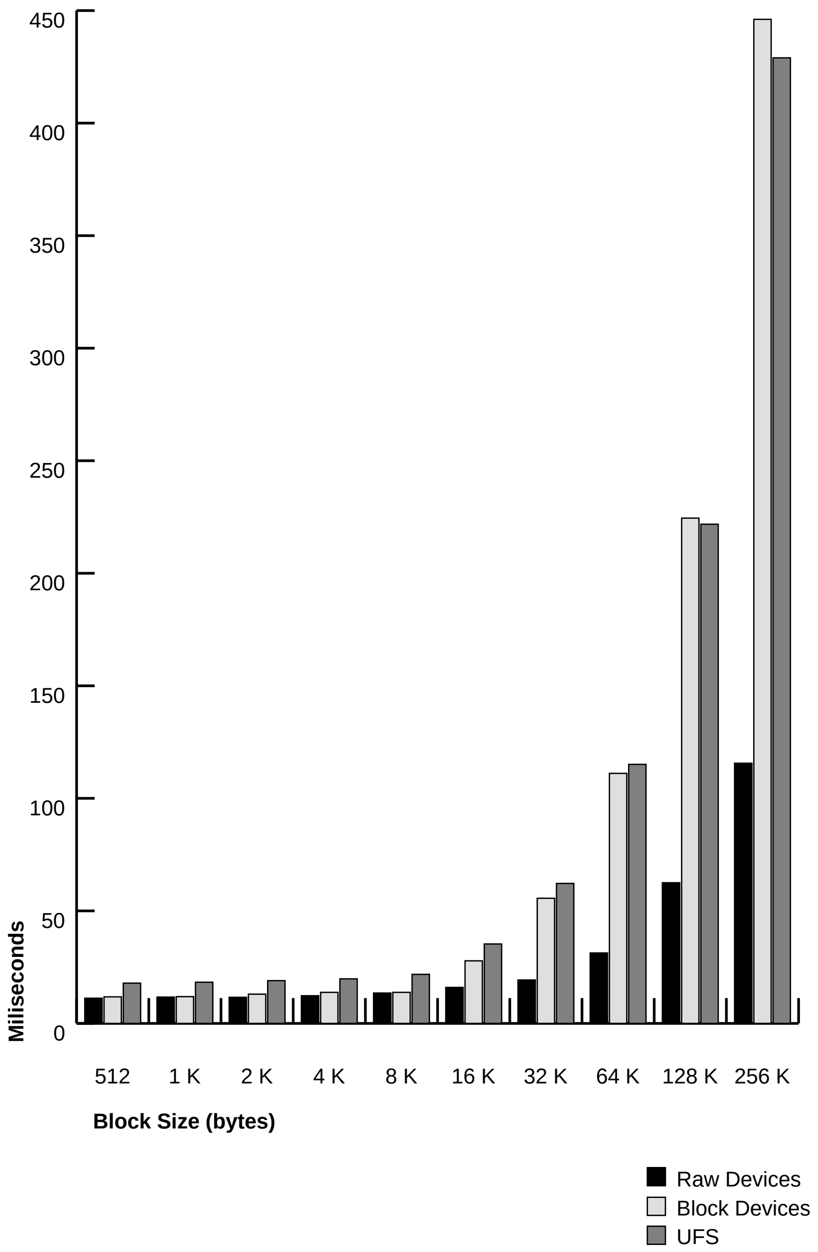
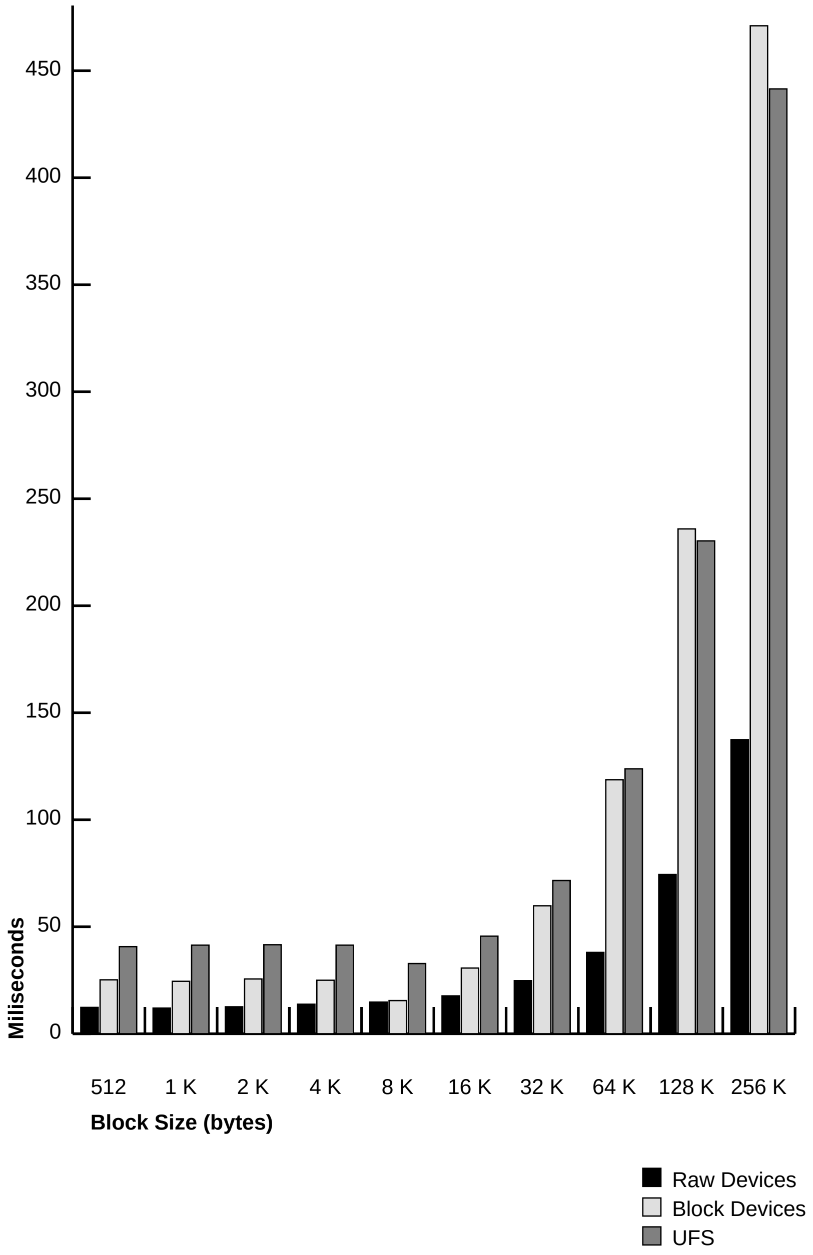


Table 20–10 and Figure 20–4 show speed of random write in milliseconds per I/O, for each of the three disk layout options on the test system.

Table 20–10 Block Size and Speed of Random Write (Sample Data)

	Speed of Random Write on		
Block Size	**Raw Device**	**Block Device**	**UNIX File System**
512 bytes	12.3	25.2	40.7
1KB	12.0	24.5	41.4
2KB	12.6	25.6	41.6
4KB	13.8	25.0	41.4
8KB	14.8	15.5	32.8
16KB	17.7	30.7	45.6
32KB	24.8	59.8	71.6
64KB	38.0	118.7	123.8
128KB	74.4	235.9	230.3
256KB	137.4	471.0	441.5

Figure 20–4 Block Size and Speed of Random Write (Sample Data)



Evaluate Disk Layout Options

Knowing the types of operation that predominate in your application and the speed with which your system can process the corresponding I/Os, you can choose the disk layout that will maximize performance.

For example, with the sample application and test system described previously, the UNIX file system would be a good choice. With random reads predominating (50% of all I/O operations), 8KB would be a good block size. Raw devices with UNIX file systems provide comparable performance of random reads at this block size. Furthermore, the UNIX file system in this example processes sequential reads (25% of all I/O operations) almost twice as fast as raw devices, given an 8KB block size.

Note: *Figures shown in the preceding example will differ significantly on different platforms, and with different disks!* To plan effectively, test I/O performance on your own system.

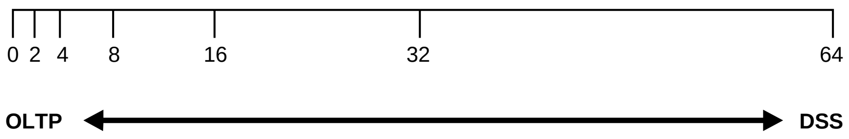
Choosing Data Block Size

Table data in the database is stored in data blocks. This section describes how to allocate space within data blocks for best performance. With single block I/O (random read), retrieve all desired data from a single block in one read for best performance. How you store the data determines whether this performance objective will be achieved. It depends on two factors: storage of the rows, and block size.

The operating system I/O size should be equal to or greater than the database block size. Sequential read performance will improve if operating system I/O size is twice or three times the database block size (as in the example in "Testing the Performance of Your Disks"). This assumes that the operating system can buffer the I/O so that the next block will be read from that particular buffer.

Figure 20–5 illustrates the suitability of various block sizes to online transaction processing (OLTP) or decision support (DSS) applications.

Figure 20–5 Block Size and Application Type



See Also: Your Oracle platform-specific documentation for information on the minimum and maximum block size on your platform.

Block Size Advantages and Disadvantages

This section describes advantages and disadvantages of different block sizes.

Table 20–11 Block Size Advantages and Disadvantages

Block Size	Advantages	Disadvantages
Small (2KB-4KB)	Reduces block contention. Good for small rows, or lots of random access.	Has relatively large overhead. You may end up storing only a small number of rows, depending on the size of the row.
Medium (8KB)	If rows are of medium size, you can bring a number of rows into the buffer cache with a single I/O. With 2KB or 4KB block size, you may only bring in a single row.	Space in the buffer cache will be wasted if you are doing random access to small rows and have a large block size. For example, with an 8KB block size and 50 byte row size, you are wasting 7,950 bytes in the buffer cache when doing random access.
Large (16KB-32KB)	There is relatively less overhead, thus more room to store useful data. Good for sequential access, or very large rows.	Large block size is not good for index blocks used in an OLTP type environment, because they increase block contention on the index leaf blocks.

Evaluating Device Bandwidth

The number of I/Os a disk can perform depends on whether the operations involve reading or writing to objects stored on raw devices or on the file system. This affects the number of disks you must use to achieve the desired level of performance.

Detecting I/O Problems

This section describes two tasks to perform if you suspect a problem with I/O usage:

- Checking System I/O Utilization
- Checking Oracle I/O Utilization

Oracle compiles file I/O statistics that reflect disk access to database files. These statistics report only the I/O utilization of Oracle sessions—yet every process affects the available I/O resources. Tuning non-Oracle factors can thus improve performance.

Checking System I/O Utilization

Use operating system monitoring tools to determine what processes are running on the system as a whole, and to monitor disk access to all files. Remember that disks holding datafiles and redo log files may also hold files that are not related to Oracle. Try to reduce any heavy access to disks that contain database files. Access to non-Oracle files can be monitored only through operating system facilities rather than through the V$FILESTAT view.

Tools such as **sar -d** on many UNIX systems enable you to examine the **iostat** I/O statistics for your entire system. (Some UNIX-based platforms have an **iostat** command.) On NT systems, use Performance Monitor.

Note: For information on other platforms, please check your operating system documentation.

Checking Oracle I/O Utilization

This section identifies the views and processes that provide Oracle I/O statistics, and shows how to check statistics using V$FILESTAT.

Which Dynamic Performance Views Contain I/O Statistics

Table 20–12 shows dynamic performance views to check for I/O statistics relating to Oracle database files, log files, archive files, and control files.

Table 20–12 Where to Find Statistics about Oracle Files

File Type	Where to Find Statistics
Database Files	V$FILESTAT
Log Files	V$SYSSTAT, V$SYSTEM_EVENT, V$SESSION_EVENT
Archive Files	V$SYSTEM_EVENT, V$SESSION_EVENT
Control Files	V$SYSTEM_EVENT, V$SESSION_EVENT

Which Processes Reflect Oracle File I/O

Table 20–13 lists processes whose statistics reflect I/O throughput for the different Oracle file types.

Table 20–13 File Throughput Statistics for Oracle Processes

	Process							
File	**LGWR**	**DBW*n***	**ARCH**	**SMON**	**PMON**	**CKPT**	**Fore-ground**	**PQ Process**
Database Files		X		X	X	X	X	X
Log Files	X							
Archive Files			X					
Control Files	X	X	X	X	X	X	X	X

V$SYSTEM_EVENT, for example, shows the total number of I/Os and average duration, by type of I/O. You can thus determine which types of I/O are too slow. If there are Oracle-related I/O problems, tune them. But if your process is not consuming the available I/O resources, then some other process is. Go back to the system to identify the process that is using up so much I/O, and determine why. Then tune this process.

Note: Different types of I/O in Oracle require different tuning approaches. Tuning I/O for data warehousing applications that perform large sequential reads is different from tuning I/O for OLTP applications that perform random reads and writes.

See Also: "Planning File Storage" on page 20-5.

Checking Oracle Datafile I/O with V$FILESTAT

Examine disk access to database files through the dynamic performance view V$FILESTAT. This view shows the following information for database I/O (but not for log file I/O):

- Number of physical reads and writes
- Number of blocks read and written
- Total I/O time for reads and writes

By default, this view is available only to the user SYS and to users granted SELECT ANY TABLE system privilege, such as SYSTEM. The following column values reflect the number of disk accesses for each datafile:

PHYRDS	The number of reads from each database file.
PHYWRTS	The number of writes to each database file.

Use the following query to monitor these values over some period of time while your application is running:

```
SELECT name, phyrds, phywrts
   FROM v$datafile df, v$filestat fs
   WHERE df.file# = fs.file#;
```

This query also retrieves the name of each datafile from the dynamic performance view V$DATAFILE. Sample output might look like this:

```
NAME                                           PHYRDS    PHYWRTS
-------------------------------------------- ---------- ----------
/oracle/ora70/dbs/ora_system.dbf                   7679       2735
/oracle/ora70/dbs/ora_temp.dbf                       32        546
```

The PHYRDS and PHYWRTS columns of V$FILESTAT can also be obtained through SNMP.

The total I/O for a single disk is the sum of PHYRDS and PHYWRTS for all the database files managed by the Oracle instance on that disk. Determine this value for each of your disks. Also determine the rate at which I/O occurs for each disk by dividing the total I/O by the interval of time over which the statistics were collected.

Solving I/O Problems

The rest of this chapter describes various techniques of solving I/O problems:

- Reducing Disk Contention by Distributing I/O
- Striping Disks
- Avoiding Dynamic Space Management
- Tuning Sorts
- Tuning Checkpoints
- Tuning LGWR and DBWn I/O
- Configuring the Large Pool

Reducing Disk Contention by Distributing I/O

This section describes how to reduce disk contention.

- What Is Disk Contention?
- Separating Datafiles and Redo Log Files
- Striping Table Data
- Separating Tables and Indexes
- Reducing Disk I/O Unrelated to Oracle

What Is Disk Contention?

Disk contention occurs when multiple processes try to access the same disk simultaneously. Most disks have limits on both the number of accesses and the amount of data they can transfer per second. When these limits are reached, processes may have to wait to access the disk.

In general, consider the statistics in the V$FILESTAT view and your operating system facilities. Consult your hardware documentation to determine the limits on the capacity of your disks. Any disks operating at or near full capacity are potential sites for disk contention. For example, 40 or more I/Os per second is excessive for most disks on VMS or UNIX operating systems.

To reduce the activity on an overloaded disk, move one or more of its heavily accessed files to a less active disk. Apply this principle to each of your disks until they all have roughly the same amount of I/O. This is referred to as *distributing I/O*.

Separating Datafiles and Redo Log Files

Oracle processes constantly access datafiles and redo log files. If these files are on common disks, there is potential for disk contention. Place each datafile on a separate disk. Multiple processes can then access different files concurrently without disk contention.

Place each set of redo log files on a separate disk with no other activity. Redo log files are written by the Log Writer process (LGWR) when a transaction is committed. Information in a redo log file is written sequentially. This sequential writing can take place much faster if there is no concurrent activity on the same disk. Dedicating a separate disk to redo log files usually ensures that LGWR runs smoothly with no further tuning attention. Performance bottlenecks related to LGWR are rare. For information on tuning LGWR, see the section "Detecting Contention for Redo Log Buffer Latches" on page 21-16.

Note: Mirroring redo log files, or maintaining multiple copies of each redo log file on separate disks, does not slow LGWR considerably. LGWR writes to each disk in parallel and waits until each part of the parallel write is complete. Because the time required for your operating system to perform a single-disk write may vary, increasing the number of copies increases the likelihood that one of the single-disk writes in the parallel write will take longer than average. A parallel write will not take longer than the longest possible single-disk write. There may also be some overhead associated with parallel writes on your operating system.

Dedicating separate disks and mirroring redo log files are important safety precautions. Dedicating separate disks to datafiles and redo log files ensures that the datafiles and the redo log files cannot both be lost in a single disk failure. Mirroring redo log files ensures that a redo log file cannot be lost in a single disk failure.

Striping Table Data

Striping, or spreading a large table’s data across separate datafiles on separate disks, can also help to reduce contention. This strategy is fully discussed in the section "Striping Disks" on page 20-22.

Separating Tables and Indexes

It is not necessary to separate a frequently used table from its index. During the course of a transaction, the index is read first, and then the table is read. Because these I/Os occur sequentially, the table and index can be stored on the same disk without contention.

Reducing Disk I/O Unrelated to Oracle

If possible, eliminate I/O unrelated to Oracle on disks that contain database files. This measure is especially helpful in optimizing access to redo log files. Not only does this reduce disk contention, it also allows you to monitor all activity on such disks through the dynamic performance table V$FILESTAT.

Striping Disks

This section describes:

- The Purpose of Striping
- I/O Balancing and Striping
- Striping Disks Manually
- Striping Disks with Operating System Software
- Striping Hardware with RAID

The Purpose of Striping

"Striping" divides a large table's data into small portions and stores these portions in separate datafiles on separate disks. This permits multiple processes to access different portions of the table concurrently without disk contention. Striping is particularly helpful in optimizing random access to tables with many rows. Striping can either be done manually (described below), or through operating system striping utilities.

I/O Balancing and Striping

Benchmark tuners in the past tried hard to ensure that the I/O load was evenly balanced across the available devices. Currently, operating systems are providing the ability to stripe a heavily used container file across many physical devices. However, such techniques are productive only where the load redistribution eliminates or reduces some form of queue.

If I/O queues exist or are suspected, then load distribution across the available devices is a natural tuning step. Where larger numbers of physical drives are available, consider dedicating two drives to carrying redo logs (two because redo logs should always be mirrored either by the operating system or using Oracle redo log group features). Because redo logs are written serially, drives dedicated to redo log activity normally require limited head movement. This significantly accelerates log writing.

When archiving, it is beneficial to use extra disks so that LGWR and ARCH do not compete for the same read/write head. This is achieved by placing logs on alternating drives.

Mirroring can also be a cause of I/O bottlenecks. The process of writing to each mirror is normally done in parallel, and does not cause a bottleneck. However, if each mirror is striped differently, then the I/O is not completed until the slowest mirror member is finished. To avoid I/O problems, stripe using the same number of disks for the destination database, or the copy, as you used for the source database.

For example, if you have 160KB of data striped over 8 disks, but the data is mirrored onto only one disk, then regardless of how quickly the data is processed on the 8 disks, the I/O is not completed until 160KB has been written onto the mirror disk. It might thus take 20.48 milliseconds to write the database, but 137 milliseconds to write the mirror.

Striping Disks Manually

To stripe disks manually, you need to relate an object's storage requirements to its I/O requirements.

1. Begin by evaluating an object's disk storage requirements by checking:

   - The size of the object
   - The size of the disk

   For example, if an object requires 5GB in Oracle storage space, you need one 5GB disk or two 4GB disks to accommodate it. On the other hand, if the system is configured with 1GB or 2GB disks, the object may require 5 or 3 disks, respectively.

2. Compare to this the application's I/O requirements, as described in "Analyzing I/O Requirements" on page 20-2. You must take the larger of the storage requirement and the I/O requirement.

   For example, if the storage requirement is 5 disks (1GB each), and the I/O requirement is 2 disks, then your application requires the higher value: 5 disks.

3. Create a tablespace with the CREATE TABLESPACE statement. Specify the datafiles in the DATAFILE clause. Each of the files should be on a different disk.

   ```
   CREATE TABLESPACE stripedtabspace
      DATAFILE ’file_on_disk_1’ SIZE 1GB,
         ’file_on_disk_2’ SIZE 1GB,
         ’file_on_disk_3’ SIZE 1GB,
         ’file_on_disk_4’ SIZE 1GB,
         ’file_on_disk_5’ SIZE 1GB;
   ```

4. Then create the table with the CREATE TABLE statement. Specify the newly created tablespace in the TABLESPACE clause.

   Also specify the size of the table extents in the STORAGE clause. Store each extent in a separate datafile. The table extents should be slightly smaller than the datafiles in the tablespace to allow for overhead. For example, when preparing for datafiles of 1GB (1024MB), you can set the table extents to be 1023MB:

   ```
   CREATE TABLE stripedtab (
      col_1  NUMBER(2),
      col_2  VARCHAR2(10) )
      TABLESPACE stripedtabspace
      STORAGE ( INITIAL 1023MB  NEXT 1023MB
         MINEXTENTS 5  PCTINCREASE 0 );
   ```

(Alternatively, you can stripe a table by entering an ALTER TABLE ALLOCATE EXTENT statement, with a DATAFILE ’*size*’ SIZE clause.)

These steps result in the creation of table STRIPEDTAB. STRIPEDTAB has 5 initial extents, each of size 1023MB. Each extent takes up one of the datafiles named in the DATAFILE clause of the CREATE TABLESPACE statement. Each of these files is on a separate disk. The 5 extents are all allocated immediately, because MINEXTENTS is 5.

See Also: *Oracle8i SQL Reference* for more information on MINEXTENTS and the other storage parameters.

Striping Disks with Operating System Software

As an alternative to striping disks manually, use operating system striping software, such as an LVM (logical volume manager), to stripe disks. With striping software, the concern is choosing the right stripe size. This depends on the Oracle block size and disk access method.

Table 20–14 Minimum Stripe Size

Disk Access	Minimum Stripe Size
Random reads and writes	The minimum stripe size is twice the Oracle block size.
Sequential reads	The minimum stripe size is twice the value of DB_FILE_MULTIBLOCK_READ_COUNT.

In striping, uniform access to the data is assumed. If the stripe size is too large, can a hot spot may appear on one disk or on a small number of disks. Avoid this by reducing the stripe size, thus spreading the data over more disks.

Consider an example in which 100 rows of fixed size are evenly distributed over 5 disks, with each disk containing 20 sequential rows. If you application only requires access to rows 35 through 55, then only 2 disks must perform the I/O. At this rate, the system cannot achieve the desired level of performance.

Correct this problem by spreading rows 35 through 55 across more disks. In the current example, if there were two rows per block, then we could place rows 35 and 36 on the same disk, and rows 37 and 38 on a different disk. Taking this approach, we could spread the data over all the disks and I/O throughput would improve.

Striping Hardware with RAID

Redundant arrays of inexpensive disks (RAID) can offer significant advantages in their failure resilience features. They also permit striping to be achieved quite easily, but do not appear to provide any significant performance advantage. In fact, they may impose a higher cost in I/O overhead.

In some instances, performance can be improved by *not* using the full features of RAID technology. In other cases, RAID technology's resilience to single component failure may justify its cost in terms of performance.

Avoiding Dynamic Space Management

When you create an object such as a table or rollback segment, Oracle allocates space in the database for the data. This space is called a *segment*. If subsequent database operations cause the data volume to increase and exceed the space allocated, Oracle extends the segment. Dynamic extension then reduces performance.

This section discusses:

- Detecting Dynamic Extension
- Allocating Extents
- Evaluating Unlimited Extents
- Evaluating Multiple Extents
- Avoiding Dynamic Space Management in Rollback Segments
- Reducing Migrated and Chained Rows
- Modifying the SQL.BSQ File

Detecting Dynamic Extension

Dynamic extension causes Oracle to execute SQL statements in addition to those SQL statements issued by user processes. These SQL statements are called *recursive calls* because Oracle issues these statements itself. Recursive calls are also generated by these activities:

- Misses on the data dictionary cache
- Firing of database triggers
- Execution of Data Definition Language statements
- Execution of SQL statements within stored procedures, functions, packages, and anonymous PL/SQL blocks
- Enforcement of referential integrity constraints

Examine the RECURSIVE CALLS statistic through the dynamic performance view V$SYSSTAT. By default, this view is available only to user SYS and to users granted the SELECT ANY TABLE system privilege, such as SYSTEM. Use the following query to monitor this statistic over a period of time:

```
SELECT name, value
    FROM v$sysstat
    WHERE NAME = 'recursive calls';
```

Oracle responds with something similar to:

```
NAME                                                          VALUE
------------------------------------------------------- ----------
recursive calls                                             626681
```

If Oracle continues to make excessive recursive calls while your application is running, determine whether these recursive calls are due to an activity, other than dynamic extension, that generates recursive calls. If you determine that the recursive calls are caused by dynamic extension, reduce this extension by allocating larger extents.

Allocating Extents

Follow these steps to avoid dynamic extension:

1. Determine the maximum size of your object. For formulas to estimate space requirements for an object, please refer to the *Oracle8i Administrator's Guide.*
2. Choose storage parameter values so Oracle allocates extents large enough to accommodate all your data when you create the object.

Larger extents tend to benefit performance for these reasons:

- Blocks in a single extent are contiguous, so one large extent is more contiguous than multiple small extents. Oracle can read one large extent from disk with fewer multiblock reads than would be required to read many small extents.
- Segments with larger extents are less likely to be extended.

However, since large extents require more contiguous blocks, Oracle may have difficulty finding enough contiguous space to store them. To determine whether to allocate only a few large extents or many small extents, evaluate the benefits and drawbacks of each in consideration of plans for the growth and use of your objects.

Automatically re-sizable datafiles can also cause problems with dynamic extension. Avoid using the automatic extension. Instead, manually allocate more space to a datafile during times when the system is relatively inactive.

Evaluating Unlimited Extents

Even though an object may have unlimited extents, this does not mean that having a large number of small extents is acceptable. For optimal performance you may decide to reduce the number of extents.

Extent maps list all extents for a particular segment. The number of extents per Oracle block depends on operating system block size and platform. Although an extent is a data structure inside Oracle, the size of this data structure depends on the platform. Accordingly, this affects the number of extents Oracle can store in a single operating system block. Typically, this value is as follows:

Table 20–15 Block Size and Maximum Number of Extents (Typical Values)

Block Size (KB)	Max. Number of Extents
2	121
4	255
8	504
16	1032
32	2070

For optimal performance, you should be able to read the extent map with a single I/O. Performance degrades if multiple I/Os are necessary for a full table scan to get the extent map.

Avoid dynamic extension in dictionary-mapped tablespaces. For dictionary-mapped tablespaces, do not let the number of extents exceed 1,000. If extent allocation is local, do not have more than 2,000 extents. Having too many extents reduces performance when dropping or truncating tables.

The optimal choice in most situations is to enable AUTOEXTEND. You can also use a proven value for allocating extents if you are sure the value provides optimal performance.

Evaluating Multiple Extents

This section explains various ramifications of using multiple extents.

- You cannot put large segments into single extents because of file size and file system size limitations. When you enable segments to allocate new extents over time, you can take advantage of faster, less expensive disks.
- For a table that is never full-table scanned, it makes no difference in terms of query performance whether the table has one extent or multiple extents.
- The performance of searches using an index is not affected by the index having one extent or multiple extents.
- Using more than one extent in a table, cluster, or temporary segment does not affect the performance of full scans on a multi-user system.
- Using more than one extent in a table, cluster, or temporary segment does not materially affect the performance of full scans on a dedicated single-user batch processing system if the extents are properly sized, and if the application is designed to avoid expensive DDL operations.
- If extent sizes are appropriately matched to the I/O size, the performance cost of having many extents in a segment will be minimized.
- For rollback segments, many extents are preferable to few extents. Having many extents reduces the number of recursive SQL calls to perform dynamic extent allocations on the segments.

Avoiding Dynamic Space Management in Rollback Segments

The size of rollback segments can affect performance. Rollback segment size is determined by the rollback segment's storage parameter values. Your rollback segments must be large enough to hold the rollback entries for your transactions. As with other objects, you should avoid dynamic space management in rollback segments.

Use the SET TRANSACTION statement to assign transactions to rollback segments of the appropriate size based on the recommendations in the following sections. If you do not explicitly assign a transaction to a rollback segment, Oracle automatically assigns it to a rollback segment.

For example, the following statement assigns the current transaction to the rollback segment OLTP_13:

```
SET TRANSACTION USE ROLLBACK SEGMENT oltp_13
```

Note: If you are running multiple concurrent copies of the same application, be careful not to assign the transactions for all copies to the same rollback segment. This leads to contention for that rollback segment.

Also monitor the shrinking, or dynamic deallocation, of rollback segments based on the OPTIMAL storage parameter. For information on choosing values for this parameter, monitoring rollback segment shrinking, and adjusting the OPTIMAL parameter, please see the *Oracle8i Administrator's Guide*.

For Long Queries

Assign large rollback segments to transactions that modify data that is concurrently selected by long queries. Such queries may require access to rollback segments to reconstruct a read-consistent version of the modified data. The rollback segments must be large enough to hold all the rollback entries for the data while the query is running.

For Long Transactions

Assign large rollback segments to transactions that modify large amounts of data. A large rollback segment can improve the performance of such a transaction, because the transaction generates large rollback entries. If a rollback entry does not fit into a rollback segment, Oracle extends the segment. Dynamic extension reduces performance and should be avoided whenever possible.

For OLTP Transactions

OLTP applications are characterized by frequent concurrent transactions, each of which modifies a small amount of data. Assign OLTP transactions to small rollback segments, provided that their data is not concurrently queried. Small rollback segments are more likely to remain stored in the buffer cache where they can be accessed quickly. A typical OLTP rollback segment might have 2 extents, each approximately 10 kilobytes in size. To best avoid contention, create many rollback segments and assign each transaction to its own rollback segment.

Reducing Migrated and Chained Rows

If an UPDATE statement increases the amount of data in a row so that the row no longer fits in its data block, Oracle tries to find another block with enough free space to hold the entire row. If such a block is available, 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, Oracle splits the row into multiple pieces and stores each piece in a separate block. This is called *chaining* a row. Rows can also be chained when they are inserted.

Dynamic space management, especially migration and chaining, is detrimental to performance:

- UPDATE statements that cause migration and chaining perform poorly
- Queries that select migrated or chained rows must perform more I/O

Identify migrated and chained rows in a table or cluster using the ANALYZE statement with the LIST CHAINED ROWS option. This statement collects information about each migrated or chained row and places this information into a specified output table.

The definition of a sample output table named CHAINED_ROWS appears in a SQL script available on your distribution medium. The common name of this script is UTLCHAIN.SQL, although its exact name and location varies depending on your platform. Your output table must have the same column names, datatypes, and sizes as the CHAINED_ROWS table.

You can also detect migrated or chained rows by checking the TABLE FETCH CONTINUED ROW column in V$SYSSTAT. Increase PCTFREE to avoid migrated rows. If you leave more free space available in the block, the row will have room to grow. You can also reorganize or re-create tables and indexes with high deletion rates.

Note: PCTUSED is not the opposite of PCTFREE; PCTUSED controls space management.

See Also: *Oracle8i Concepts* for more information.

To reduce migrated and chained rows in an existing table, follow these steps:

1. Use the ANALYZE statement to collect information about migrated and chained rows. For example:

   ```
   ANALYZE TABLE order_hist LIST CHAINED ROWS;
   ```

2. Query the output table:

```
SELECT *
   FROM chained_rows
   WHERE table_name = 'ORDER_HIST';

OWNER_NAME  TABLE_NAME  CLUST... HEAD_ROWID          TIMESTAMP
----------  ----------  -----... ------------------  ---------
SCOTT       ORDER_HIST       ... AAAAluAAHAAAAA1AAA  04-MAR-96
SCOTT       ORDER_HIST       ... AAAAluAAHAAAAA1AAB  04-MAR-96
SCOTT       ORDER_HIST       ... AAAAluAAHAAAAA1AAC  04-MAR-96
```

The output lists all rows that are either migrated or chained.

3. If the output table shows that you have many migrated or chained rows, you can eliminate migrated rows with the following steps:

   a. Create an intermediate table with the same columns as the existing table to hold the migrated and chained rows:

   ```
   CREATE TABLE int_order_hist
      AS SELECT *
         FROM order_hist
         WHERE ROWID IN
            (SELECT head_rowid
               FROM chained_rows
               WHERE table_name = 'ORDER_HIST');
   ```

   b. Delete the migrated and chained rows from the existing table:

   ```
   DELETE FROM order_hist
      WHERE ROWID IN
         (SELECT head_rowid
            FROM chained_rows
            WHERE table_name = 'ORDER_HIST');
   ```

   c. Insert the rows of the intermediate table into the existing table:

   ```
   INSERT INTO order_hist
      SELECT *
         FROM int_order_hist;
   ```

   d. Drop the intermediate table:

   ```
   DROP TABLE int_order_history;
   ```

4. Delete the information collected in step 1 from the output table:

   ```
   DELETE FROM chained_rows
      WHERE table_name = 'ORDER_HIST';
   ```

5. Use the ANALYZE statement again and query the output table.
6. Any rows that appear in the output table are chained. You can eliminate chained rows only by increasing your data block size. It may not be possible to avoid chaining in all situations. Chaining is often unavoidable with tables that have a LONG column or long CHAR or VARCHAR2 columns.

Retrieval of migrated rows is resource intensive; therefore, all tables subject to UPDATE should have their distributed free space set to allow enough space within the block for the likely update.

Modifying the SQL.BSQ File

The SQL.BSQ file runs when you issue the CREATE DATABASE statement. This file contains the table definitions that make up the Oracle server. The views you use as a DBA are based on these tables. Oracle recommends that you strictly limit modifications to SQL.BSQ.

- If necessary, you can increase the value of the following storage parameters: INITIAL, NEXT, MINEXTENTS, MAXEXTENTS, PCTINCREASE, FREELISTS, FREELIST GROUPS, and OPTIMAL.
- With the exception of PCTINCREASE, do not decrease the setting of a storage parameter to a value below the default. (If the value of MAXEXTENTS is large, you can lower the value for PCTINCREASE or even set it to zero.)
- No other changes to SQL.BSQ are supported. In particular, you should not add, drop, or rename a column.

Note: Oracle may add, delete, or change internal data dictionary tables from release to release. For this reason, modifications you make are not carried forward when the dictionary is migrated to later releases.

See Also: *Oracle8i SQL Reference* for complete information about these parameters.

Tuning Sorts

There is a trade-off between performance and memory usage. For best performance, most sorts should occur in memory; sorts written to disk adversely affect performance. If the sort area size is too large, too much memory may be used. If the sort area size is too small, sorts may have to be written to disk which, as, mentioned, can severely degrade performance.

This section describes:

- Sorting to Memory
- Sorting to Disk
- Optimizing Sort Performance with Temporary Tablespaces
- Using NOSORT to Create Indexes Without Sorting
- GROUP BY NOSORT

Sorting to Memory

The default sort area size is adequate to hold all the data for most sorts. However, if your application often performs large sorts on data that does not fit into the sort area, you may want to increase the sort area size. Large sorts can be caused by any SQL statement that performs a sort on many rows.

> **See Also:** *Oracle8i Concepts* lists SQL statements that perform sorts.

Recognizing Large Sorts

Oracle collects statistics that reflect sort activity and stores them in the dynamic performance view V$SYSSTAT. By default, this view is available only to the user SYS and to users granted the SELECT ANY TABLE system privilege. These statistics reflect sort behavior:

SORTS(MEMORY)	The number of sorts small enough to be performed entirely in sort areas without I/O to temporary segments on disk.
SORTS(DISK)	The number of sorts too large to be performed entirely in the sort area, requiring I/O to temporary segments on disk.

Use the following query to monitor these statistics over time:

```
SELECT name, value
   FROM v$sysstat
   WHERE name IN ('sorts (memory)', 'sorts (disk)');
```

The output of this query might look like this:

```
NAME                                                       VALUE
------------------------------------------------------- ----------
sorts(memory)                                                965
sorts(disk)                                                    8
```

The information in V$SYSSTAT can also be obtained through the SNMP (Simple Network Management Protocol).

Increasing SORT_AREA_SIZE to Avoid Sorting to Disk

SORT_AREA_SIZE is a dynamically modifiable initialization parameter that specifies the maximum amount of memory to use for each sort. If a significant number of sorts require disk I/O to temporary segments, your application's performance may benefit from increasing the size of the sort area. In this case, increase the value of SORT_AREA_SIZE.

The maximum value of this parameter depends on your operating system. You need to determine how large a SORT_AREA_SIZE makes sense. If you set SORT_AREA_SIZE to an adequately large value, most sorts should not have to go to disk (unless, for example, you are sorting a 10-gigabyte table).

> **See Also:** "Tuning Checkpoints" on "Tuning Checkpoints" and "SORT_AREA_SIZE" on page 26-22.

Performance Benefits of Large Sort Areas

As mentioned, increasing sort area size decreases the chances that sorts go to disk. Therefore, with a larger sort area, most sorts will process quickly without I/O.

When Oracle writes sort operations to disk, it writes out partially sorted data in sorted runs. Once all the data has been received by the sort, Oracle merges the runs to produce the final sorted output. If the sort area is not large enough to merge all the runs at once, subsets of the runs are merged in several merge passes. If the sort area is larger, there will be fewer, longer runs produced. A larger sort area also means the sort can merge more runs in one merge pass.

Performance Trade-offs for Large Sort Areas

Increasing sort area size causes each Oracle sort process to allocate more memory. This increase reduces the amount of memory for private SQL and PL/SQL areas. It can also affect operating system memory allocation and may induce paging and swapping. Before increasing the size of the sort area, be sure enough free memory is available on your operating system to accommodate a larger sort area.

If you increase sort area size, consider decreasing the value for the SORT_AREA_RETAINED_SIZE parameter. This parameter controls the lower limit to which Oracle reduces the size of the sort area when Oracle completes some or all of a sort process. That is, Oracle reduces the size of the sort area once the sort has started sending the sorted data to the user or to the next part of the query. A smaller retained sort area reduces memory usage but causes additional I/O to write and read data to and from temporary segments on disk.

Sorting to Disk

If you sort to disk, make sure that PCTINCREASE is set to zero for the tablespace used for sorting. Also, INITIAL and NEXT should be the same size. This reduces fragmentation of the tablespaces used for sorting. You set these parameters using the STORAGE option of ALTER TABLE.

> **See Also:** *Oracle8i Concepts* for more information on PCTINCREASE.

Optimizing Sort Performance with Temporary Tablespaces

Optimize sort performance by performing sorts in temporary tablespaces. To create temporary tablespaces, use the CREATE TABLESPACE or ALTER TABLESPACE statements with the TEMPORARY keyword.

Normally, a sort may require many space allocation calls to allocate and deallocate temporary segments. If you specify a tablespace as TEMPORARY, Oracle caches one sort segment in that tablespace for each instance requesting a sort operation. This scheme bypasses the normal space allocation mechanism and greatly improves performance of medium-sized sorts that cannot be done completely in memory.

You cannot use the TEMPORARY keyword with tablespaces containing permanent objects such as tables or rollback segments.

> **See Also:** *Oracle8i SQL Reference* for more information about the syntax of the CREATE TABLESPACE and ALTER TABLESPACE statements.

Improving Sort Performance by Striping Temporary Tablespaces

Stripe the temporary tablespace over many disks, preferably using an operating system striping tool. For example, if you only stripe the temporary tablespace over 2 disks with a maximum of 50 I/Os per second on each disk, then Oracle can only perform 100 I/Os per second. This restriction could lengthen the duration of sort operations.

For the previous example, you could accelerate sort processing fivefold if you striped the temporary tablespace over 10 disks. This would enable 500 I/Os per second.

Improving Sort Performance Using SORT_MULTIBLOCK_READ_COUNT

Another way to improve sort performance using temporary tablespaces is to tune the parameter SORT_MULTIBLOCK_READ_COUNT. For temporary segments, SORT_MULTIBLOCK_READ_COUNT has nearly the same effect as the parameter DB_FILE_MULTIBLOCK_READ_COUNT.

Increasing the value of SORT_MULTIBLOCK_READ_COUNT forces the sort process to read a larger section of each sort run from disk to memory during each merge pass. This also forces the sort process to reduce the merge width, or number of runs, that can be merged in one merge pass. This may increase in the number of merge passes.

Because each merge pass produces a new sort run to disk, an increase in the number of merge passes causes an increase in the total amount of I/O performed during the sort. Carefully balance increases in I/O throughput obtained by increasing the SORT_MULTIBLOCK_READ_COUNT parameter with possible increases in the total amount of I/O performed.

Using NOSORT to Create Indexes Without Sorting

One cause of sorting is the creation of indexes. Creating an index for a table involves sorting all rows in the table based on the values of the indexed columns. Oracle also allows you to create indexes without sorting. If the rows in the table are loaded in ascending order, you can create the index faster without sorting.

The NOSORT Option

To create an index without sorting, load the rows into the table in ascending order of the indexed column values. Your operating system may provide a sorting utility to sort the rows before you load them. When you create the index, use the NOSORT option on the CREATE INDEX statement. For example, this CREATE INDEX statement creates the index EMP_INDEX on the ENAME column of the EMP table without sorting the rows in the EMP table:

```
CREATE INDEX emp_index
   ON emp(ename)
   NOSORT;
```

Note: Specifying NOSORT in a CREATE INDEX statement negates the use of PARALLEL INDEX CREATE, even if PARALLEL (DEGREE *n*) is specified.

When to Use the NOSORT Option

Presorting your data and loading it in order may not always be the fastest way to load a table.

- If you have a multiple-CPU computer, you may be able to load data faster using multiple processors in parallel, each processor loading a different portion of the data. To take advantage of parallel processing, load the data without sorting it first. Then create the index *without* the NOSORT option.
- If you have a single-CPU computer, you should sort your data before loading, if possible. Then create the index *with* the NOSORT option.

GROUP BY NOSORT

Sorting can be avoided when performing a GROUP BY operation when you know that the input data is already ordered so that all rows in each group are clumped together. This may be the case, for example, if the rows are being retrieved from an index that matches the grouped columns, or if a sort-merge join produces the rows in the right order. ORDER BY sorts can be avoided in the same circumstances. When no sort takes place, the EXPLAIN PLAN output indicates GROUP BY NOSORT.

Tuning Checkpoints

A checkpoint is an operation that Oracle performs automatically. This section explains:

- How Checkpoints Affect Performance
- Choosing Checkpoint Frequency
- Fast-Start Checkpointing

How Checkpoints Affect Performance

Checkpoints affect:

- Instance recovery time performance
- Run-time performance

Frequent checkpoints can reduce instance recovery time in the event of an instance failure. If checkpoints are relatively frequent, then relatively few changes to the database are made between checkpoints. In this case, relatively few changes must be rolled forward for instance recovery.

Checkpoints can momentarily reduce run-time performance because checkpoints cause DBW*n* processes to perform I/O. However, the overhead associated with checkpoints is usually small and affects performance only while Oracle performs the checkpoint.

Choosing Checkpoint Frequency

Choose a checkpoint frequency based on your performance concerns. If you are more concerned with efficient run-time performance than recovery time, choose a lower checkpoint frequency. If you are more concerned with having fast instance recovery than with achieving optimal run-time performance, choose a higher checkpoint frequency.

Because checkpoints are necessary for redo log maintenance, you cannot eliminate checkpoints entirely. However, you can reduce checkpoint frequency to a minimum by setting these parameters:

- Set the value of the LOG_CHECKPOINT_INTERVAL initialization parameter (in multiples of physical block size) to be larger than the size of your largest redo log file.
- Set the value of the LOG_CHECKPOINT_TIMEOUT initialization parameter to zero. This value eliminates time-based checkpoints.

You can also control performance by setting a limit on the number of I/O operations as described under the following heading, "Fast-Start Checkpointing".

In addition to setting these parameters, also consider the size of your log files. Maintaining small log files can increase checkpoint activity and reduce performance.

Note: Oracle checkpoints current read blocks. By contrast, sort blocks and consistent read blocks are *not* checkpointed.

See Also: *Oracle8i Concepts* for a complete discussion of checkpoints.

Fast-Start Checkpointing

The Fast-Start Checkpointing feature limits the number of dirty buffers and thereby limits the amount of time required for instance recovery. If Oracle must process an excessive number of I/O operations to perform instance recovery, performance can be adversely affected. You can control this overhead by setting an appropriate value for the parameter FAST_START_IO_TARGET.

FAST_START_IO_TARGET limits the number of I/O operations that Oracle should allow for instance recovery. If the number of operations required for recovery at any point in time exceeds this limit, Oracle writes dirty buffers to disk until the number of I/O operations needed for instance recovery is reduced to the limit set by FAST_START_IO_TARGET.

You can control the duration of instance recovery because the number of operations required to recover indicates how much time this process takes. Disable this aspect of checkpointing by setting FAST_START_IO_TARGET to zero (0).

Note: Fast-Start Checkpointing is only available with the Oracle8*i* Enterprise Edition.

Tuning LGWR and DBWn I/O

This section describes how to tune I/O for the log writer and database writer background processes:

- Tuning LGWR I/O
- Tuning DBWn I/O

Tuning LGWR I/O

Applications with many INSERTs or with LONG/RAW activity may benefit from tuning LGWR I/O. The size of each I/O write depends on the size of the log buffer which is set by the initialization parameter LOG_BUFFER. It is thus important to choose the right log buffer size. LGWR starts writing if the buffer is one third full, or when it is posted by a foreground process such as a COMMIT. Too large a log buffer size might delay the writes. Too small a log buffer might also be inefficient, resulting in frequent, small I/Os.

If the average size of the I/O becomes quite large, the log file could become a bottleneck. To avoid this problem, you can stripe the redo log files, going in parallel to several disks. You must use an operating system striping tool, because manual striping is not possible in this situation.

Stripe size is likewise important. You can figure an appropriate value by dividing the average redo I/O size by the number of disks over which you want to stripe the buffer.

If you have a large number of datafiles or are in a high OLTP environment, you should always have the CHECKPOINT_PROCESS initialization parameter set to TRUE. This setting enables the CKPT process, ensuring that during a checkpoint LGWR keeps on writing redo information, while the CKPT process updates the datafile headers.

Incremental Checkpointing

Incremental checkpointing improves the performance of crash and instance recovery, but not media recovery. An incremental checkpoint records the position in the redo thread (log) from which crash/instance recovery needs to begin. This log position is determined by the oldest dirty buffer in the buffer cache. The incremental checkpoint information is maintained periodically with minimal or no overhead during normal processing.

The duration of instance recovery is most influenced by the number of data blocks the recovery process must read from disk. You can control the duration of recovery processing using the parameter DB_BLOCK_MAX_DIRTY_TARGET. This parameter allows you to restrict the number of blocks the instance recovery process must read from disk during recovery.

To set an appropriate value for DB_BLOCK_MAX_DIRTY_TARGET, first determine how long your system takes to read one block from disk. Then divide your desired instance recovery period duration by this value. For example, if it takes 10 milliseconds to read one block and you do not want your recovery process to last longer than 30 seconds, set the value for DB_BLOCK_MAX_DIRTY_TARGET to 3000. The reduced instance recovery time achieved by setting DB_BLOCK_MAX_DIRTY_TARGET to a lower value is obtained at the cost of more writes during normal processing.

Setting this parameter to a smaller value imposes higher overhead during normal processing because Oracle must write more buffers to disk. On the other hand, the smaller the value of this parameter, the better the recovery performance, because fewer blocks need to be recovered. You can also use DB_BLOCK_MAX_DIRTY_TARGET to limit the number of blocks read during instance recovery and thus influence the duration of recovery processing.

Incremental checkpoint information is maintained automatically by Oracle without affecting other checkpoints, such as user-specified checkpoints. In other words, incremental checkpointing occurs independently of other checkpoints occurring in the instance.

Incremental checkpointing is beneficial for recovery in a single instance as well as a multi-instance environment.

> **See Also:** *Oracle8i Concepts, Oracle8i Reference* and the *Oracle8i Backup and Recovery Guide.*

Tuning DBWn I/O

This section describes the following issues of tuning DBW I/O:

- Multiple Database Writer (DBWn) Processes
- Internal Write Batch Size
- LRU Latches with a Single Buffer Pool
- LRU Latches with Multiple Buffer Pools

Multiple Database Writer (DBWn) Processes

Using the DB_WRITER_PROCESSES initialization parameter, you can create multiple database writer processes (from DBW0 to DBW9). These may be useful for high-end systems such as NUMA machines and SMP systems that have a large number of CPUs. These background processes are not the same as the I/O server processes (set with DBWR_IO_SLAVES); the latter can die without the instance failing. You cannot concurrently run I/O server processes and multiple DBW*n* processes on the same system.

Internal Write Batch Size

Database writer (DBW*n)* process(es) use the *internal write batch size,* which is set to the *lowest* of the following three values (A, B, or C):

- Value A is calculated as follows:

$$\frac{DB\_FILES * DB\_FILE\_SIMULTANEOUS\_WRITES}{2} = Value\ A$$

- Value B is the port-specific limit. (See your Oracle platform-specific documentation.)
- Value C is one-fourth the value of DB_BLOCK_BUFFERS.

Setting the internal write batch size too large may result in uneven response times.

For best results, you can influence the internal write batch size by changing the parameter values by which Value A in the formula above is calculated. Take the following approach:

1. Determine the files to which you must write, and the number of disks on which those files reside.
2. Determine the number of I/Os you can perform against these disks.
3. Determine the number of writes that your transactions require.
4. Make sure you have enough disks to sustain this rate.

LRU Latches with a Single Buffer Pool

When you have multiple database writer (DBW*n*) processes and only one buffer pool, the buffer cache is divided up among the processes by LRU (least recently used) latches; each LRU latch is for one LRU list.

The default value of the DB_BLOCK_LRU_LATCHES parameter is the number of CPUs in the system. You can adjust this value to be equal to, or a multiple of, the number of CPUs. The objective is to cause each DBW*n* process to have the same number of LRU lists, so that they have equivalent loads.

For example, if you have 2 database writer processes and 4 LRU lists (and thus 4 latches), the DBW*n* processes obtain latches in a round-robin fashion. DBW0 obtains latch 1, DBW1 obtains latch 2, then DBW2 obtains latch 3 and DBW3 obtains latch 4. Similarly, if your system has 8 CPUs and 3 DBW*n* processes, you should have 9 latches.

LRU Latches with Multiple Buffer Pools

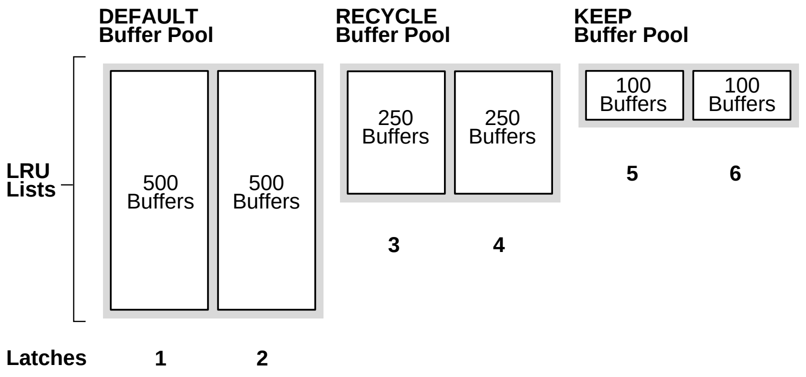
However, if you are using multiple buffer pools and multiple database writer (DBW*n*) processes, the number of latches in each pool (DEFAULT, KEEP, and

RECYCLE) should be equal to, or a multiple of, the number of processes. This is recommended so that each DBW*n* process will be equally loaded.

Note: When there are multiple buffer pools, each buffer pool has a contiguous range of LRU latches.

Consider the example in Figure 20–6 where there are 3 DBW*n* processes and 2 latches for each of the 3 buffer pools, for a total of 6 latches. Each buffer pool would obtain a latch in round robin fashion.

Figure 20–6 LRU Latches with Multiple Buffer Pools: Example 1



The DEFAULT buffer pool has 500 buffers for each LRU list. The RECYCLE buffer pool has 250 buffers for each LRU list. The KEEP buffer pool has 100 buffers for each LRU.

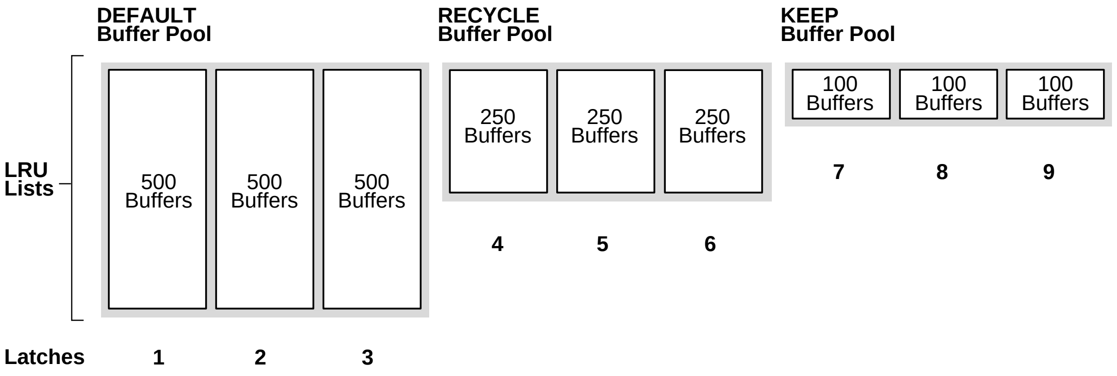
DBW0 gets latch 1 (500) and latch 4 (250) for 750.
DBW1 gets latch 2 (500) and latch 6 (100) for 600.
DBW2 gets latch 3 (250) and latch 5 (100) for 350.

Thus the load carried by each of the DBW*n* processes differs, and performance suffers. If, however, there are 3 latches in each pool, the DBW*n* processes have equal loads and performance is optimized.

The different buffer pools have different rates of block replacement. Ordinarily, blocks are rarely modified in the KEEP pool and frequently modified in the RECYCLE pool; which means you need to write out blocks more frequently from the RECYCLE pool than from the KEEP pool. As a result, owning 100 buffers from one pool is not the same as owning 100 buffers from the other pool. To be perfectly load balanced, each DBW*n* process should have the same number of LRU lists from each type of buffer pool.

A well configured system might have 3 DBW*n* processes and 9 latches, with 3 latches in each buffer pool

Figure 20–7 LRU Latches with Multiple Buffer Pools: Example 2



The DEFAULT buffer pool has 500 buffers for each LRU list. The RECYCLE buffer pool has 250 buffers for each LRU list. The KEEP buffer pool has 100 buffers for each LRU list.

DBW0 gets latch 1 (500) and latch 4 (250) and latch 7 (100) for 850.
DBW1 gets latch 2 (500) and latch 5 (250) and latch 8 (100) for 850.
DBW2 gets latch 3 (500) and latch 6 (250) and latch 9 (100) for 850.

Tuning Backup and Restore Operations

The primary goal of backup and restore tuning is create an adequate flow of data between disk and storage device. Tuning backup and restore operations requires that you complete the following tasks:

- Locating the Source of a Bottleneck
- Using Fixed Views to Monitor Backup Operations
- Improving Backup Throughput

Locating the Source of a Bottleneck

Typically, you perform backups and restore operations in three phases:

- Reading the input (disk or tape)
- Processing data by validating blocks and copying them from the input to the output buffer
- Writing the output to tape or disk

It is unlikely that all phases take the same amount of time. Therefore, the slowest of the three phases is the bottleneck.

Understanding the Types of I/O

Oracle backup and restore uses two types of I/O: disk and tape. When performing a backup, the input files are read using disk I/O, and the output backup file is written using either disk or tape I/O. When performing restores, these roles reverse. Both disk and tape I/O can be synchronous or asynchronous; each is independent of the other.

Measuring Synchronous and Asynchronous I/O Rates

When using synchronous I/O, you can easily determine how much time backup jobs require because devices only perform one I/O task at a time. When using asynchronous I/O, it is more difficult to measure the bytes-per-second rate, for the following reasons:

- Asynchronous processing implies that more than one task occurs at a time.
- Oracle I/O uses a polling rather than an interrupt mechanism to determine when each I/O request completes. Because the backup or restore process is not immediately notified of I/O completion by the operating system, you cannot determine the duration of each I/O.

The following sections explain how to use the V$BACKUP_SYNC_IO and V$BACKUP_ASYNC_IO views to determine the bottleneck in a backup.

> **See Also:** For more information about these views, please refer to the *Oracle8i Reference.*

Determining Bottlenecks with Synchronous I/O

With synchronous I/O, it is difficult to identify specific bottlenecks because all synchronous I/O is a bottleneck to the process. The only way to tune synchronous I/O is to compare the bytes-per-second rate with the device's maximum throughput rate. If the bytes-per-second rate is lower than that device specifies, consider tuning that part of the backup/restore process. Use the V$BACKUP_SYNC_IO.DISCRETE_BYTES_PER_SECOND column to see the I/O rate.

Determining Bottlenecks with Asynchronous I/O

If the combination of LONG_WAITS and SHORT_WAITS is a significant fraction of IO_COUNT, then the file indicated in V$BACKUP_SYNCH_IO and V$BACKUP_ASYNCH_IO is probably a bottleneck. Some platforms' implementation of asynchronous I/O can cause the caller to wait for I/O completion when performing a non-blocking poll for I/O. Because this behavior can vary among platforms, the V$BACKUP_ASYNC_IO view shows the total time for both "short" and " long" waits.

"Long" waits are the number of times the backup/restore process told the operating system to wait until an I/O was complete. "Short" waits are the number of times the backup/restore process made an operating system call to poll for I/O completion in a non-blocking mode. Both types of waits the operating system should respond immediately.

If the SHORT_WAIT_TIME_TOTAL column is equal to or greater than the LONG_WAIT_TIME_TOTAL column, then your platform probably blocks for I/O completion when performing "non-blocking" I/O polling. In this case, the SHORT_WAIT_TIME_TOTAL represents real I/O time for this file. If the SHORT_WAIT_TIME_TOTAL is low compared to the total time for this file, then the delay is most likely caused by other factors, such as process swapping. If possible, tune your operating system so the I/O wait time appears up in the LONG_WAIT_TIME_TOTAL column.

Using Fixed Views to Monitor Backup Operations

Use V$BACKUP_SYNC_IO and V$BACKUP_ASYNC_IO to determine the source of backup or restore bottlenecks and to determine the progress of backup jobs.

V$BACKUP_SYNC_IO contains rows when the I/O is synchronous to the process (or "thread," on some platforms) performing the backup. V$BACKUP_ASYNC_IO contains rows when the I/O is asynchronous. Asynchronous I/O is obtained either with I/O processes or because it is supported by the underlying operating system.

Columns Common to V$BACKUP_SYNC_IO and V$BACKUP_ASYNC_IO

Table 20–16 lists columns and their descriptions that are common to the V$BACKUP_SYNC_IO and V$BACKUP_ASYNC_IO views.

Table 20–16 Common Columns of V$BACKUP_SYNC_IO and V$BACKUP_ASYNC_IO

Column	Description
SID	Oracle SID of the session doing the backup or restore.
SERIAL	Usage counter for the SID doing the backup or restore.
USE_COUNT	A counter you can use to identify rows from different backup sets. Each time a new set of rows is created in V$BACKUP_SYNC_IO OR V$BACKUP_ASYNC_IO, they have a USE_COUNT that is greater than the previous rows. The USE_COUNT is the same for all rows used by each backup or restore operation.
DEVICE_TYPE	Device type where the file is located (typically DISK or SBT_TAPE).
TYPE	INPUT: The file(s) are being read.
	OUTPUT: The file(s) are being written.
	AGGREGATE: This row represents the total I/O counts for all DISK files involved in the backup or restore.
STATUS	NOT STARTED: This file has not been opened yet.
	IN PROGRESS: This file is currently being read or written.
	FINISHED: Processing for this file is complete.
FILENAME	The name of the backup file being read or written.
SET_COUNT	The SET_COUNT of the backup set being read or written.
SET_STAMP	The SET_STAMP of the backup set being read or written.
BUFFER_SIZE	Size of the buffers being used to read/write this file in bytes.
BUFFER_COUNT	The number of buffers being used to read/write this file.
TOTAL_BYTES	The total number of bytes to be read or written for this file if known. If not known, this column is null.
OPEN_TIME	Time this file was opened. If TYPE = 'AGGREGATE', then this is the time that the first file in the aggregate was opened.
CLOSE_TIME	Time this file was closed. If TYPE = 'AGGREGATE', then this is the time that the last file in the aggregate was closed.

Table 20–16   Common Columns of V$BACKUP_SYNC_IO and V$BACKUP_ASYNC_IO

Column	Description
ELAPSED_TIME	The length of time expressed in 100ths of seconds that the file was open.
MAXOPENFILES	The number of concurrently open DISK files. This value is only present in rows where TYPE = 'AGGREGATE'.
BYTES	The number of bytes read or written so far.
EFFECTIVE_BYTES_PER_SECOND	The I/O rate achieved with this device during the backup. It is the number of bytes read or written divided by the elapsed time. This value is only meaningful for the component of the backup system causing a bottleneck. If this component is not causing a bottleneck, then the speed measured by this column actually reflects the speed of some other, slower, component of the system.
IO_COUNT	The number of I/Os performed to this file. Each request is to read or write one buffer, of size BUFFER_SIZE.

Columns Specific to V$BACKUP_SYNC_IO

Table 20–17 lists columns specific to the V$BACKUP_SYNC_IO view.

Table 20–17   Columns Specific to V$BACKUP_SYNC_IO

Column	Description
IO_TIME_TOTAL	The total time required to perform I/O for this file expressed in 100ths of seconds.
IO_TIME_MAX	The maximum time taken for a single I/O request.
DISCRETE_BYTES_PER_SECOND	The average transfer rate for this file. This is based on measurements taken at the start and end of each individual I/O request. This value should reflect the real speed of this device.

Columns Specific to V$BACKUP_ASYNC_IO

Table 20–18 lists columns specific to the V$BACKUP_ASYNC_IO view.

Table 20–18 Columns Specific to V$BACKUP_ASYNC_IO

Column	Description
READY	The number of asynchronous requests for which a buffer was immediately ready for use.
SHORT_WAITS	The number of times a buffer was not immediately available, but then a buffer became available after doing a non-blocking poll for I/O completion. The reason non-blocking waits are timed is because some implementations of "asynchronous I/O" may wait for an I/O to complete even when the request is supposed to be non-blocking.
SHORT_WAIT_TIME_TOTAL	The total time expressed in 100ths of seconds, taken by non-blocking polls for I/O completion.
SHORT_WAIT_TIME_MAX	The maximum time taken for a non-blocking poll for I/O completion, in 100ths of seconds.
LONG_WAITS	The number of times a buffer was not immediately available, and only became available after issuing a blocking wait for an I/O to complete.
LONG_WAIT_TIME_TOTAL	The total time expressed in 100ths of seconds taken by blocking waits for I/O completion.
LONG_WAIT_TIME_MAX	The maximum time taken for a blocking wait for I/O completion expressed in 100ths of seconds.

Improving Backup Throughput

In optimally tuned backups, tape components should create the only bottleneck. You should keep the tape and its device "streaming", or constantly rotating. If the tape is not streaming, the data flow to the tape may be inadequate.

This section contains the following topics to maintain streaming by improving backup throughput:

- Understanding Factors Affecting Data Transfer Rates
- Determining Whether the Tape is Streaming for Synchronous I/O
- Determining Whether the Tape is Streaming for Asynchronous I/O
- Increasing Throughput to Enable Tape Streaming

Understanding Factors Affecting Data Transfer Rates

The rate at which the host sends data to keep the tape streaming depends on these factors:

- The raw capacity of the tape device
- Compression

Tape device raw capacity is the *smallest* amount of data required to keep the tape streaming.

Compression is implemented either in the tape hardware or by the media management software. If you do not use compression, then the raw capacity of the tape device keeps it streaming. If you use compression, then the amount of data that must be sent to stream the tape is the raw device capacity multiplied by the compression factor. The compression factor varies for different types of data.

Determining Whether the Tape is Streaming for Synchronous I/O

To determine whether your tape is streaming when the I/O is synchronous, query the EFFECTIVE_BYTES_PER_SECOND column in the V$BACKUP_SYNC_IO view:

If EFFECTIVE_BYTES_PER_SECOND is	Then
Less than the raw capacity of the hardware	The tape is not streaming.
More than the raw capacity of the hardware	The tape may be streaming, depending on the compression ratio of the data.

Determining Whether the Tape is Streaming for Asynchronous I/O

If the I/O is asynchronous, the tape is streaming if the combination of LONG_WAITS and SHORT_WAITS is a significant fraction of I/O count. Place more importance on SHORT_WAITS if the time indicated in the SHORT_WAIT_TIME_TOTAL column is equal or greater than the LONG_WAIT_TIME_TOTAL column.

Increasing Throughput to Enable Tape Streaming

If the tape is not streaming, the basic strategy is to supply more bytes-per-second to the tape. Modify this strategy depending on the how many blocks Oracle must read from disk and how many disks Oracle must access.

Spreading I/O Across Multiple Disks Using the DISKRATIO parameter of the BACKUP statement to distribute backup I/O across multiple volumes, specify how many disk drives RMAN uses to distribute file reads when backing up multiple concurrent datafiles. For example, assume that your system uses 10 disks, the disks supply data at 10 byes/second, and the tape drive requires 50 bytes/second to keep streaming. In this case, set DISKRATIO equal to 5 to spread the backup load onto 5 disks.

When setting DISKRATIO, spread the I/O over only as many disks as needed to keep the tape streaming: any more can increase the time it would take to restore a single file and provides no performance benefit. Note that if you do not specify DISKRATIO but specify FILESPERSET, DISKRATIO defaults to FILESPERSET. If neither is specified, DISKRATIO defaults to 4.

Backing Up Empty Files or Files with Few Changes When performing a full backup of files that are largely empty or performing an incremental backup when few blocks have changed, you may not be able to supply data to the tape fast enough to keep it streaming.

In this case, achieve optimal performance by using:

- The highest possible value for the MAXOPENFILES parameter of the Recovery Manager SET LIMIT CHANNEL statement
- Asynchronous disk I/O

The latter takes advantage of asynchronous read-ahead that fills input buffers from one file while processing data from others.

See Also: For more information about the RMAN SET statement, see the *Oracle8i Backup and Recovery Guide*.

Backing Up Full Files When you perform a full backup of files that are mostly full and the tape is not streaming, you can improve performance in several ways as shown in Table 20–19:

Table 20–19 Throughput performance improvement methods

Method	Consequence
Set BACKUP_TAPE_IO_SLAVES	Simulates asynchronous tape I/O by spawning multiple processes to divide the work for the backup or restore operation. If you do not set this parameter, then all I/O to the tape layer is synchronous. If you set this parameter, set LARGE_POOL_SIZE as well.
Set LARGE_POOL_SIZE	If you set BACKUP_TAPE_IO_SLAVES, then the buffers for tape I/O must be allocated from shared memory so they can be shared between two processes. Oracle does the following when attempting to get shared buffers for I/O slaves: ▪ If LARGE_POOL_SIZE is set, Oracle attempts to get memory from the large pool. If this value is not large enough, then Oracle does not try to get buffers from the shared pool. ▪ If LARGE_POOL_SIZE is not set, Oracle attempts to get memory from the shared pool. ▪ If Oracle cannot get enough memory, then it obtains I/O buffer memory from local process memory and writes a message to the alert.log file indicating that synchronous I/O will be used for this backup.
Increase DB_FILE_DIRECT_IO_COUNT	Causes RMAN to use larger buffers for disk I/O. The default buffer size used for backup and restore disk I/O is DB_FILE_DIRECT_IO_COUNT * DB_BLOCK_SIZE. The default value for DB_FILE_DIRECT_IO_COUNT is 64, so if DB_BLOCK_SIZE is 2048, then the buffer size is 128KB. On some platforms, the most efficient I/O buffer size may be more than 128KB.
Make sure the RMAN parameters MAXOPENFILES or FILESPERSET are not too low	Increases the number of files that RMAN can process at one time. Using default buffer sizes, each concurrently open file uses 512KB of process memory (or SGA large pool memory, if I/O processes are used) for buffers. The number of concurrent files should be just enough to keep the tape streaming. You must derive the correct number by trial and error because unused block compression greatly affects the amount of disk data that is sent to the tape drive. If your tape drives are slower than your disk drives, then a value of 1 for MAXOPENFILES should be sufficient.

Configuring the Large Pool

You can optionally configure the large pool so Oracle has a separate pool from which it can request large memory allocations. This prevents competition with other subsystems for the same memory.

As Oracle allocates more shared pool memory for the multi-threaded server session memory, the amount of shared pool memory available for the shared SQL cache decreases. If you allocate session memory from another area of shared memory, Oracle can use the shared pool primarily for caching shared SQL and not incur the performance overhead from shrinking the shared SQL cache.

For I/O server processes and backup and restore operations, Oracle allocates buffers that are a few hundred kilobytes in size. Although the shared pool may be unable to satisfy this request, the large pool will be able to do so. The large pool does not have an LRU list; Oracle will not attempt to age memory out of the large pool.

Use the LARGE_POOL_SIZE parameter to configure the large pool. To see in which pool (shared pool or large pool) the memory for an object resides, see the column POOL in V$SGASTAT.

> **See Also:** *Oracle8i Concepts* for further information about the large pool and the *Oracle8i Reference* for complete information about initialization parameters.

21 Tuning Resource Contention

Contention occurs when multiple processes try to access the same resource simultaneously. Some processes must then wait for access to various database structures. Topics discussed in this chapter include:

Understanding Contention Issues

Symptoms of resource contention problems can be found in V$SYSTEM_EVENT. This view reveals various system problems that may be impacting performance, problems such as latch contention, buffer contention, and I/O contention. It is important to remember that these are only *symptoms* of problems—not the actual causes.

For example, by looking at V$SYSTEM_EVENT you might notice lots of buffer-busy waits. It may be that many processes are inserting into the same block and must wait for each other before they can insert. The solution might be to introduce free lists for the object in question.

Buffer busy waits may also have caused some latch free waits. Because most of these waits were caused by misses on the cache buffer hash chain latch, this was also a side effect of trying to insert into the same block. Rather than increasing SPINCOUNT to reduce the latch free waits (a symptom), you should change the object to allow for multiple processes to insert into free blocks. This approach will effectively reduce contention.

See Also: *Oracle8i Administrator's Guide* to understand which resources are used by various Oracle features.

Detecting Contention Problems

The V$RESOURCE_LIMIT view provides information about current and maximum global resource utilization for some system resources. This information enables you to make better decisions when choosing values for resource limit-controlling parameters.

If the system has idle time, start your investigation by checking V$SYSTEM_EVENT. Examine the events with the highest average wait time, then take appropriate action on each. For example, if you find a high number of latch free waits, look in V$LATCH to see which latch is the problem.

For excessive buffer busy waits, look in V$WAITSTAT to see which block type has the highest wait count and the highest wait time. Look in V$SESSION_WAIT for cache buffer waits so you can decode the file and block number of an object.

The rest of this chapter describes common contention problems. Remember that the different forms of contention are symptoms which can be fixed by making changes in one of two places:

- Changes in the application
- CHANGES in Oracle

Sometimes you have no alternative but to change the application in order to overcome performance constraints.

Solving Contention Problems

The rest of this chapter examines various kinds of contention and explains how to resolve problems. Contention may be for rollback segments, multi-threaded server processes, parallel execution servers, redo log buffer latches, LRU latch, or for free lists.

Reducing Contention for Rollback Segments

In this section, you will learn how to reduce contention for rollback segments. The following issues are discussed:

- Identifying Rollback Segment Contention
- Creating Rollback Segments

Identifying Rollback Segment Contention

Contention for rollback segments is reflected by contention for buffers that contain rollback segment blocks. You can determine whether contention for rollback segments is adversely affecting performance by checking the dynamic performance table V$WAITSTAT.

V$WAITSTAT contains statistics that reflect block contention. By default, this table is available only to the user SYS and to other users who have SELECT ANY TABLE system privilege, such as SYSTEM. These statistics reflect contention for different classes of block:

SYSTEM UNDO HEADER	the number of waits for buffers containing header blocks of the SYSTEM rollback segment
SYSTEM UNDO BLOCK	the number of waits for buffers containing blocks of the SYSTEM rollback segment other than header blocks
UNDO HEADER	the number of waits for buffers containing header blocks of rollback segments other than the SYSTEM rollback segment
UNDO BLOCK	the number of waits for buffers containing blocks other than header blocks of rollback segments other than the SYSTEM rollback segment

Use the following query to monitor these statistics over a period of time while your application is running:

```
SELECT CLASS, COUNT
   FROM V$WAITSTAT
   WHERE CLASS IN ('SYSTEM UNDO HEADER', 'SYSTEM UNDO BLOCK',
      'UNDO HEADER', 'UNDO BLOCK');
```

The result of this query might look like this:

```
CLASS                   COUNT
------------------ ----------
SYSTEM UNDO HEADER       2089
SYSTEM UNDO BLOCK         633
UNDO HEADER              1235
UNDO BLOCK                942
```

Compare the number of waits for each class of block with the total number of requests for data over the same period of time. You can monitor the total number of requests for data over a period of time with this query:

```
SELECT SUM(VALUE)
   FROM V$SYSSTAT
   WHERE NAME IN ('DB BLOCK GETS', 'CONSISTENT GETS');
```

The output of this query might look like this:

```
SUM(VALUE)
----------
   929530
```

The information in V$SYSSTAT can also be obtained through SNMP.

If the number of waits for any class of block exceeds 1% of the total number of requests, consider creating more rollback segments to reduce contention.

Creating Rollback Segments

To reduce contention for buffers containing rollback segment blocks, create more rollback segments. Table 21–1 shows some general guidelines for choosing how many rollback segments to allocate based on the number of concurrent transactions on your database. These guidelines are appropriate for most application mixes.

Table 21–1 Choosing the Number of Rollback Segments

Number of Current Transactions (*n*)	**Number of Rollback Segments Recommended**
n < 16	4
16 <= n < 32	8
32 <= n	n / 4

Reducing Contention for Multi-threaded Server Processes

In this section, you will learn how to reduce contention for processes used by Oracle's multi-threaded server architecture:

- Reducing Contention for Dispatcher Processes
- Reducing Contention for Shared Server Processes

Identifying Contention Using the Dispatcher-specific Views

The following views provide dispatcher performance statistics:

- V$DISPATCHER
- V$DISPATCHER_RATE

V$DISPATCHER provides general information about dispatcher processes. V$DISPATCHER_RATE view provides dispatcher processing statistics.

> **See Also:** For detailed information about these views, please refer to the *Oracle8i Reference*.

Analyzing V$DISPATCHER_RATE Statistics

The V$DISPATCHER_RATE view contains current, average, and maximum dispatcher statistics for several categories. Statistics with the prefix "CUR_" are statistics for the current session. Statistics with the prefix "AVG_" are the average values for the statistics since the collection period began. Statistics with "MAX_" prefixes are the maximum values for these categories since statistics collection began.

To assess dispatcher performance, query the V$DISPATCHER_RATE view and compare the current values with the maximums. If your present system throughput provides adequate response time and current values from this view are near the average and below the maximum, you likely have an optimally tuned MTS environment.

If the current and average rates are significantly below the maximums, consider reducing the number of dispatchers. Conversely, if current and average rates are close to the maximums, you may need to add more dispatchers. A good rule-of-thumb is to examine V$DISPATCHER_RATE statistics during both light and heavy system use periods. After identifying your MTS load patterns, adjust your parameters accordingly.

If needed, you can also mimic processing loads by running system stress-tests and periodically polling the V$DISPATCHER_RATE statistics. Proper interpretation of these statistics varies from platform to platform. Different types of applications also can cause significant variations on the statistical values recorded in V$DISPATCHER_RATE.

Reducing Contention for Dispatcher Processes

This section discusses how to identify contention for dispatcher processes, how to add dispatcher processes, and how to enable connection pooling.

Identifying Contention for Dispatcher Processes

Contention for dispatcher processes is indicated by either of these symptoms:

- Excessive busy rates for existing dispatcher processes
- Steady increases in waiting times for responses in the response queues of dispatcher processes

Examining Busy Rates for Dispatcher Processes V$DISPATCHER contains statistics reflecting the activity of dispatcher processes. By default, this view is available only to the user SYS and to other users who have SELECT ANY TABLE system privilege, such as SYSTEM. These columns reflect busy rates for dispatcher processes:

IDLE	Displays the idle time for the dispatcher process in hundredths of a second
BUSY	Displays the busy time for the dispatcher process in hundredths of a second

Use the following query to monitor these statistics over a period of time while your application is running:

```
SELECT NETWORK "PROTOCOL",
      SUM(BUSY) / ( SUM(BUSY) + SUM(IDLE) )  "TOTAL BUSY RATE"
   FROM V$DISPATCHER
  GROUP BY NETWORK;
```

This query returns the total busy rate for the dispatcher processes of each protocol; that is, the percentage of time the dispatcher processes of each protocol are busy. The result of this query might look like this:

```
PROTOCOL  TOTAL BUSY RATE
--------  ---------------
DECNET         .004589828
TCP            .029111042
```

From this result, you can make these observations:

- DECnet dispatcher processes are busy nearly 0.5% of the time
- TCP dispatcher processes are busy nearly 3% of the time

If the database is only in use 8 hours per day, statistics need to be normalized by the effective work times. You cannot simply look at statistics from the time the instance started. Instead, record statistics during peak workloads. If the dispatcher processes for a specific protocol are busy for more than 50% of the peak workload period, then by adding dispatcher processes you may improve performance for users connected to Oracle using that protocol.

Examining Wait Times for Dispatcher Process Response Queues V$QUEUE contains statistics reflecting the response queue activity for dispatcher processes. By default, this table is available only to the user SYS and to other users who have SELECT ANY TABLE system privilege, such as SYSTEM. These columns show wait times for responses in the queue:

WAIT	the total waiting time, in hundredths of a second, for all responses that have ever been in the queue
TOTALQ	the total number of responses that have ever been in the queue

Use the following query to monitor these statistics occasionally while your application is running:

```
SELECT NETWORK     "PROTOCOL",
   DECODE( SUM(TOTALQ), 0, ’NO RESPONSES’,
      SUM(WAIT)/SUM(TOTALQ) || ’ HUNDREDTHS OF SECONDS’)
   "AVERAGE WAIT TIME PER RESPONSE"
   FROM V$QUEUE Q, V$DISPATCHER D
   WHERE Q.TYPE = ’DISPATCHER’
      AND Q.PADDR = D.PADDR
   GROUP BY NETWORK;
```

This query returns the average time, in hundredths of a second, that a response waits in each response queue for a dispatcher process to route it to a user process. This query uses the V$DISPATCHER table to group the rows of the V$QUEUE table by network protocol. The query also uses the DECODE syntax to recognize those protocols for which there have been no responses in the queue. The result of this query might look like this:

```
PROTOCOL  AVERAGE WAIT TIME PER RESPONSE
--------  ------------------------------
DECNET    .1739130 HUNDREDTHS OF SECONDS
TCP       NO RESPONSES
```

From this result, you can tell that a response in the queue for DECNET dispatcher processes waits an average of 0.17 hundredths of a second and that there have been no responses in the queue for TCP dispatcher processes.

If the average wait time for a specific network protocol continues to increase steadily as your application runs, then by adding dispatcher processes you may be able to improve performance of those user processes connected to Oracle using that protocol.

Adding Dispatcher Processes

Add dispatcher processes while Oracle is running by using the SET option of the ALTER SYSTEM command to increase the value for the MTS_DISPATCHERS parameter.

The total number of dispatcher processes across all protocols is limited by the value of the initialization parameter MTS_MAX_DISPATCHERS. You may need to increase this value before adding dispatcher processes. The default value of this parameter is 5 and the maximum value varies depending on your operating system.

See Also: *Oracle8i Administrator's Guide* for more information on adding dispatcher processes.

Enabling Connection Pooling

When system load increases and dispatcher throughput is maximized, it is not necessarily a good idea to immediately add more dispatchers. Instead, consider configuring the dispatcher to support more users with multiplexing. To do this, you must install connection manager software.

MTS_DISPATCHERS lets you enable various attributes for each dispatcher. Oracle supports a name-value syntax to let you specify attributes in a position-independent, case-insensitive manner. For example:

```
MTS_DISPATCHERS = "(PROTOCOL=TCP)(DISPATCHERS=3)(POOL=ON)(TICK=1)"
```

The optional attribute POOL (or POO) is used to enable the Net8 connection pooling feature. TICK is the size of a network TICK in seconds. The TICK - default is 15 seconds.

See Also: For more information about the MTS_DISPATCHER parameter and its options, please refer to the *Oracle8i SQL Reference* and the *Net8 Administrator's Guide*.

Enabling Connection Multiplexing

Multiplexing uses a Connection Manager (CM) process to establish and maintain connections from multiple users to individual dispatcher processes. For example, several user processes may connect to one dispatcher process by way of a single CM process.

The CM manages communication from users to the dispatcher by way of the single connection. At any one time, zero, one, or a few users may need the connection while other user processes linked to the dispatcher by way of the CM process are idle. In this way, multiplexing is beneficial as it maximizes use of user-to-dispatcher process connections.

The CM process may be on the same machine as the user and dispatcher processes or the CM process may be on a different one. Regardless, your platform's network protocol serves as the communication link between the user processes, the CM, and the dispatcher processes.

The limit on the number of connections for each dispatcher is platform dependent. Oracle recommends allocating no more than 250 connections per dispatcher. For most 32-bit machines, performance tends to degrade if the number of connections exceeds 250.

Reducing Contention for Shared Server Processes

This section discusses how to identify contention for shared server processes and increase the maximum number of shared server processes.

Identifying Contention for Shared Server Processes

Steadily increasing wait times in the requests queue indicate contention for shared server processes. To examine wait time data, use the dynamic performance view V$QUEUE. This view contains statistics showing request queue activity for shared server processes. By default, this view is available only to the user SYS and to other users with SELECT ANY TABLE system privilege, such as SYSTEM. These columns show wait times for requests in the queue:

WAIT	Displays the total waiting time, in hundredths of a second, for all requests that have ever been in the queue.
TOTALQ	Displays he total number of requests that have ever been in the queue.

Monitor these statistics occasionally while your application is running by issuing the following SQL statement:

```
SELECT DECODE( totalq, 0, 'No Requests',
      WAIT/TOTALQ || ' HUNDREDTHS OF SECONDS')
"AVERAGE WAIT TIME PER REQUESTS"
FROM V$QUEUE
WHERE TYPE = 'COMMON';
```

This query returns the results of a calculation that shows the following:

```
AVERAGE WAIT TIME PER REQUEST
-----------------------------
.090909 HUNDREDTHS OF SECONDS
```

From the result, you can tell that a request waits an average of 0.09 hundredths of a second in the queue before processing.

You can also determine how many shared server processes are currently running by issuing this query:

```
SELECT COUNT(*) "Shared Server Processes"
   FROM v$shared_servers
  WHERE status != 'QUIT';
```

The result of this query might look like this:

```
SHARED SERVER PROCESSES
-----------------------
10
```

If you detect resource contention with MTS, first make sure your LARGE_POOL_SIZE parameter allocates 5KB for each user connecting through MTS. If performance remains poor, you may want to create more resources to reduce shared server process contention. Do this by modifying the optional server process parameters as explained under the following headings.

> **See Also:** For more information about LARGE_POOL_SIZE, refer to the *Oracle8i Reference*.

Setting and Modifying MTS Processes

This section explains how to set optional parameters affecting processes for the multi-threaded server architecture. This section also explains how and when to modify these parameters to tune performance.

The static initialization parameters discussed in this section are:

- MTS_MAX_DISPATCHERS
- MTS_MAX_SERVERS

This section also describes the initialization/session parameters:

- MTS_DISPATCHERS
- MTS_SERVERS

Static Dispatcher and Server Parameters Values for the init.ora parameters MTS_MAX_DISPATCHERS and MTS_MAX_SERVERS define upper limits for the number of dispatchers and servers running on an instance. These parameters are static and cannot be changed once your database is running. You can create as many dispatcher and server processes as you need, but the total number of processes cannot exceed the host operating system's limit for the number of running processes.

Note: Setting MTS_MAX_DISPATCHERS sets the limit on dispatchers for all network protocols combined.

Dynamic Dispatcher and Server Parameters You can also define starting values for the number of dispatchers and servers by setting the MTS_DISPATCHERS and MTS_SERVERS parameters. After system startup, you can dynamically re-set values for these parameters to change the number of dispatchers and servers using the SET option of the ALTER SYSTEM command. If you enter values for these parameters in excess of limits set by the static parameters, Oracle uses the static parameter values.

Static and Dynamic MTS Parameter Dependencies The default value of MTS_MAX_SERVERS is dependent on the value of MTS_SERVERS. If MTS_SERVERS is less than or equal to 10, MTS_MAX_SERVERS defaults to 20. If MTS_SERVERS is greater than 10, MTS_MAX_SERVERS defaults to 2 times the value of MTS_SERVERS.

Self-adjusting MTS Architecture Features When the database starts, the number of MTS_SERVERS is equal to the number of shared servers. This number also serves as the minimal limit; the number of shared servers never falls below this minimum. During processing, Oracle automatically adds shared server processes up to the limit defined by MTS_MAX_SERVERS if loads on existing processes increase drastically. Therefore, you are unlikely to improve performance by explicitly adding shared server processes. However, you may need to adjust your system to accommodate certain resource issues.

If the number of shared server processes has reached the limit set by the initialization parameter MTS_MAX_SERVERS and the average wait time in the request queue is still unacceptable, you might improve performance by increasing the MTS_MAX_SERVERS value.

If resource demands exceed expectations, you can either allow Oracle to automatically add shared server processes or you can add shared processes by altering the value for MTS_SERVERS. You can change the value of this parameter in the INIT.ORA file, or alter it using the MTS_SERVERS parameter of the ALTER SYSTEM command. Experiment with this limit and monitor shared servers to determine an ideal setting for this parameter.

Determining the Optimal Number of Dispatchers and Shared Server Processes

As mentioned, MTS_SERVERS determines the number of shared server processes activated at instance startup. The default setting for MTS_SERVERS is 1 which is the default setting when MTS_DISPATCHERS is also activated.

To determine the optimal number of dispatchers and shared servers, consider the number of users typically accessing the database and how much processing each requires. Also consider that user and processing loads vary over time. For example, a customer service system's load might vary drastically from peak OLTP-oriented daytime use to DSS-oriented nighttime use. System use can also predictably change over longer time periods such as the loads experienced by an accounting system that vary greatly from mid-month to month-end.

If each user makes relatively few requests over a given period of time, then each associated user process is idle for a large percentage of time. In this case, one shared server process can serve 10 to 20 users. If each user requires a significant amount of processing, establish a higher ratio of server processes to user processes.

In the beginning, it is best to allocate fewer shared server processes. Additional shared servers start automatically as needed and are deallocated automatically if they remain idle too long. However, the initial servers always remain allocated, even if they are idle.

If you set the initial number of servers too high, your system might incur unnecessary overhead. Experiment with the number of initial shared server processes and monitor shared servers until you achieve ideal system performance for your typical database activity.

Estimating the Maximum Number of Dispatcher Processes Use values for MTS_MAX_DISPATCHERS and MTS_DISPATCHERS that are at least equal to the maximum number of concurrent sessions divided by the number of connections per dispatcher. For most systems, a value of 250 connections per dispatcher provides good performance.

Disallowing Further MTS Use with Concurrent MTS Use As mentioned, you can use the SET option of the ALTER SYSTEM command to alter the number of active, shared server processes. To prevent additional users from accessing shared server processes, set MTS_SERVERS to 0. This temporarily disables additional use of MTS. Re- setting MTS_SERVERS to a positive value enables MTS for all current users.

See Also: For information about dispatchers, see the description of the V$DISPATCHER and V$DISPATCHER_RATE views in the *Oracle8i Reference*. For more information about the ALTER SYSTEM command, see the *Oracle8i SQL Reference*. For more information on changing the number of shared servers, see the *Oracle8i Administrator's Guide*.

Reducing Contention for Parallel Execution Servers

This section describes how to detect and alleviate contention for parallel execution servers when using parallel execution:

- Identifying Contention for Parallel Execution Servers
- Reducing Contention for Parallel Execution Servers

Identifying Contention for Parallel Execution Servers

Statistics in the V$PQ_SYSSTAT view are useful for determining the appropriate number of parallel execution servers for an instance. The statistics that are particularly useful are SERVERS BUSY, SERVERS IDLE, SERVERS STARTED, and SERVERS SHUTDOWN.

Frequently, you will not be able to increase the maximum number of parallel execution servers for an instance because the maximum number is heavily dependent upon the capacity of your CPUs and your I/O bandwidth. However, if servers are continuously starting and shutting down, you should consider increasing the value of the initialization parameter PARALLEL_MIN_SERVERS.

For example, if you have determined that the maximum number of concurrent parallel execution servers that your machine can manage is 100, you should set PARALLEL_MAX_SERVERS to 100. Next, determine how many parallel execution servers the average parallel operation needs, and how many parallel operations are likely to be executed concurrently. For this example, assume you will have two concurrent operations with 20 as the average degree of parallelism. Thus at any given time there could be 80 parallel execution servers busy on an instance. Thus you should set the PARALLEL_MIN_SERVERS parameter to 80.

Periodically examine V$PQ_SYSSTAT to determine whether the 80 parallel execution servers for the instance are actually busy. To do so, issue the following query:

```
SELECT * FROM V$PQ_SYSSTAT
WHERE STATISTIC = "SERVERS BUSY";
```

The result of this query might look like this:

```
STATISTIC             VALUE
--------------------- -----------
SERVERS BUSY          70
```

Reducing Contention for Parallel Execution Servers

If you find that typically there are fewer than PARALLEL_MIN_SERVERS busy at any given time, your idle parallel execution servers constitute system overhead that is not being used. Consider decreasing the value of the parameter PARALLEL_MIN_SERVERS. If you find that there are typically more parallel execution servers active than the value of PARALLEL_MIN_SERVERS and the SERVERS STARTED statistic is continuously growing, consider increasing the value of the parameter PARALLEL_MIN_SERVERS.

Reducing Contention for Redo Log Buffer Latches

Contention for redo log buffer access rarely inhibits database performance. However, Oracle provides methods to monitor and reduce any latch contention that does occur. This section covers:

- Detecting Contention for Space in the Redo Log Buffer
- Detecting Contention for Redo Log Buffer Latches
- Examining Redo Log Activity
- Reducing Latch Contention

Detecting Contention for Space in the Redo Log Buffer

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

The statistic REDO BUFFER ALLOCATION RETRIES reflects the number of times a user process waits for space in the redo log buffer. This statistic is available through the dynamic performance view V$SYSSTAT. By default, this view is available only to the user SYS and to users granted SELECT ANY TABLE system privilege, such as SYSTEM.

Use the following query to monitor these statistics over a period of time while your application is running:

```
SELECT NAME, VALUE
   FROM V$SYSSTAT
   WHERE NAME = 'REDO BUFFER ALLOCATION RETRIES';
```

The information in V$SYSSTAT can also be obtained through the Simple Network Management Protocol (SNMP).

The value of REDO BUFFER ALLOCATION RETRIES should be near zero. If this value increments consistently, processes have had to wait for space in the buffer. The wait may be caused by the log buffer being too small or by checkpointing. Increase the size of the redo log buffer, if necessary, by changing the value of the initialization parameter LOG_BUFFER. The value of this parameter, expressed in bytes, must be a multiple of DB_BLOCK_SIZE. Alternatively, improve the checkpointing or archiving process.

Note: Multiple archiver processes are not recommended. A single automatic ARCH process can archive redo logs, keeping pace with the LGWR process.

Detecting Contention for Redo Log Buffer Latches

Access to the redo log buffer is regulated by two types of latches: the redo allocation latch and redo copy latches.

The Redo Allocation Latch

The redo allocation latch controls the allocation of space for redo entries in the redo log buffer. To allocate space in the buffer, an Oracle user process must obtain the redo allocation latch. Because there is only one redo allocation latch, only one user process can allocate space in the buffer at a time. The single redo allocation latch enforces the sequential nature of the entries in the buffer.

After allocating space for a redo entry, the user process may copy the entry into the buffer. This is called "copying on the redo allocation latch". A process may only copy on the redo allocation latch if the redo entry is smaller than a threshold size.

Redo Copy Latches

The user process first obtains the copy latch which allows the process to copy. Then it obtains the allocation latch, performs allocation, and releases the allocation latch.

Next the process performs the copy under the copy latch, and releases the copy latch. The allocation latch is thus held for only a very short period of time, as the user process does not try to obtain the copy latch while holding the allocation latch.

If the redo entry is too large to copy on the redo allocation latch, the user process must obtain a redo copy latch before copying the entry into the buffer. While holding a redo copy latch, the user process copies the redo entry into its allocated space in the buffer and then releases the redo copy latch.

If your computer has multiple CPUs, your redo log buffer can have multiple redo copy latches. These allow multiple processes to concurrently copy entries to the redo log buffer concurrently.

On single-CPU computers, there should be no redo copy latches, because only one process can be active at once. In this case, all redo entries are copied on the redo allocation latch, regardless of size.

Examining Redo Log Activity

Heavy access to the redo log buffer can result in contention for redo log buffer latches. Latch contention can reduce performance. Oracle collects statistics for the activity of all latches and stores them in the dynamic performance view V$LATCH. By default, this table is available only to the user SYS and to other users who have SELECT ANY TABLE system privilege, such as SYSTEM.

Each row in the V$LATCH table contains statistics for a different type of latch. The columns of the table reflect activity for different types of latch requests. The distinction between these types of requests is whether the requesting process continues to request a latch if it is unavailable:

WILLING-TO-WAIT	If the latch requested with a willing-to-wait request is not available, the requesting process waits a short time and requests the latch again. The process continues waiting and requesting until the latch is available.
IMMEDIATE	If the latch requested with an immediate request is not available, the requesting process does not wait, but continues processing.

These columns of the V$LATCH view reflect willing-to-wait requests:

GETS	Shows the number of successful willing-to-wait requests for a latch
MISSES	Shows the number of times an initial willing-to-wait request was unsuccessful
SLEEPS	Shows the number of times a process waited and requested a latch after an initial willing-to-wait request

For example, consider the case in which a process makes a willing-to-wait request for a latch that is unavailable. The process waits and requests the latch again and the latch is still unavailable. The process waits and requests the latch a third time and acquires the latch. This activity increments the statistics as follows:

- The GETS value increases by one because one request for the latch (the third request) was successful.
- The MISSES value increases by one each time because the initial request for the latch resulted in waiting.
- The SLEEPS value increases by two because the process waited for the latch twice, once after the initial request and again after the second request.

These columns of the V$LATCH table reflect immediate requests:

IMMEDIATE GETS	This column shows the number of successful immediate requests for each latch.
IMMEDIATE MISSES	This column shows the number of unsuccessful immediate requests for each latch.

Use the following query to monitor the statistics for the redo allocation latch and the redo copy latches over a period of time:

```
SELECT ln.name, gets, misses, immediate_gets, immediate_misses
   FROM v$latch l, v$latchname ln
   WHERE ln.name IN ('redo allocation', 'redo copy')
     AND ln.latch# = l.latch#;
```

The output of this query might look like this:

```
NAME                     GETS       MISSES     IMMEDIATE_GETS  IMMEDIATE_MISSES
------------------------ ---------- ---------- --------------- ----------------
redo allocation          252867     83         0                0
redo copy                0          0          22830            0
```

From the output of the query, calculate the wait ratio for each type of request.

Contention for a latch may affect performance if either of these conditions is true:

- If the ratio of MISSES to GETS exceeds 1%
- If the ratio of IMMEDIATE_MISSES to the sum of IMMEDIATE_GETS and IMMEDIATE_MISSES exceeds 1%

If either of these conditions is true for a latch, try to reduce contention for that latch. These contention thresholds are appropriate for most operating systems, though some computers with many CPUs may be able to tolerate more contention without performance reduction.

Reducing Latch Contention

Most cases of latch contention occur when two or more Oracle processes concurrently attempt to obtain the same latch. Latch contention rarely occurs on single-CPU computers, where only a single process can be active at once.

Reducing Contention for the Redo Allocation Latch

To reduce contention for the redo allocation latch, you should minimize the time that any single process holds the latch. To reduce this time, reduce copying on the redo allocation latch. Decreasing the value of the LOG_SMALL_ENTRY_MAX_SIZE initialization parameter reduces the number and size of redo entries copied on the redo allocation latch.

Reducing Contention for Redo Copy Latches

On multiple-CPU computers, multiple redo copy latches allow multiple processes to copy entries to the redo log buffer concurrently. The default value of LOG_SIMULTANEOUS_COPIES is the number of CPUs available to your Oracle instance.

If you observe contention for redo copy latches, add more latches by increasing the value of LOG_SIMULTANEOUS_COPIES. It can help to have up to twice as many redo copy latches as CPUs available to your Oracle instance.

Reducing Contention for the LRU Latch

The LRU (least recently used) latch controls the replacement of buffers in the buffer cache. For symmetric multiprocessor (SMP) systems, Oracle automatically sets the number of LRU latches to a value equal to one half the number of CPUs on the system. For non-SMP systems, one LRU latch is sufficient.

Contention for the LRU latch can impede performance on SMP machines with a large number of CPUs. You can detect LRU latch contention by querying V$LATCH, V$SESSION_EVENT, and V$SYSTEM_EVENT. To avoid contention, consider bypassing the buffer cache or redesigning the application.

You can specify the number of LRU latches on your system with the initialization parameter DB_BLOCK_LRU_LATCHES. This parameter sets the maximum value for the desired number of LRU latches. Each LRU latch controls a set of buffers; Oracle balances allocation of replacement buffers among the sets.

To select the appropriate value for DB_BLOCK_LRU_LATCHES, consider the following:

- The maximum number of latches is twice the number of CPUs in the system. That is, the value of DB_BLOCK_LRU_LATCHES can range from 1 to twice the number of CPUs.
- A latch should have no less than 50 buffers in its set; for small buffer caches there is no added value if you select a larger number of sets. The size of the buffer cache determines a maximum boundary condition on the number of sets.
- Do not create multiple latches when Oracle runs in *single process* mode. Oracle automatically uses only one LRU latch in single process mode.
- If the workload on the instance is large, then you should have a higher number of latches. For example, if you have 32 CPUs in your system, choose a number between half the number of CPUs (16) and actual number of CPUs (32) in your system.

Note: You cannot dynamically change the number of sets during the lifetime of the instance.

Reducing Free List Contention

Free list contention can reduce the performance of some applications. This section covers:

- Identifying Free List Contention
- Adding More Free Lists

Identifying Free List Contention

Contention for free lists is reflected by contention for free data blocks in the buffer cache. You can determine whether contention for free lists is reducing performance by querying the dynamic performance view V$WAITSTAT.

> **See Also:** For information on free lists, please refer to *Oracle8i Concepts*,

The V$WAITSTAT table contains block contention statistics. By default, this view is available only to the user SYS and to other users who have SELECT ANY TABLE system privilege, such as SYSTEM.

Use the following procedure to find the segment names and free lists that have contention:

1. Check V$WAITSTAT for contention on DATA BLOCKS.
2. Check V$SYSTEM_EVENT for BUFFER BUSY WAITS.

   High numbers indicate that some contention exists.
3. In this case, check V$SESSION_WAIT to see, for each buffer busy wait, the values for FILE, BLOCK, and ID.
4. Construct a query as follows to obtain the name of the objects and free lists that have the buffer busy waits:

   ```
   SELECT SEGMENT_NAME, SEGMENT_TYPE
   FROM DBA_EXTENTS
   WHERE FILE_ID = file
   AND BLOCK BETWEEN block_id AND block_id + blocks;
   ```

   This will return the segment name (*segment)* and type (*type)*.
5. To find the free lists, query as follows:

   ```
   SELECT SEGMENT_NAME, FREELISTS
   ```

```
FROM DBA_SEGMENTS
WHERE SEGMENT_NAME = SEGMENT
AND SEGMENT_TYPE = TYPE;
```

Adding More Free Lists

To reduce contention for the free lists of a table, re-create the table with a larger value for the FREELISTS storage parameter. Increasing the value of this parameter to the number of Oracle processes that concurrently insert data into the table may improve performance of the INSERT statements.

Re-creating the table may simply involve dropping and creating it again. However, you may instead want to use one of these methods:

- Re-create the table by selecting data from the old table into a new table, dropping the old table, and renaming the new one.
- Use Export and Import to export the table, drop the table, and import the table. This measure avoids consuming space by creating a temporary table.

22 Tuning Networks

This chapter introduces networking issues that affect tuning. Topics in this chapter include

- Detecting Network Problems
- Solving Network Problems

Detecting Network Problems

Networks entail overhead that adds a certain amount of delay to processing. To optimize performance, you must ensure that your network throughput is fast, and that you reduce the number of messages that must be sent over the network.

It can be difficult to measure the delay the network adds. Three dynamic performance views are useful for this task: V$SESSION_EVENT, V$SESSION_WAIT, and V$SESSTAT.

In V$SESSION_EVENT, the AVERAGE_WAIT column indicates the amount of time that Oracle waits between messages. You can use this statistic as a yardstick to evaluate the effectiveness of the network.

In V$SESSION_WAIT, the EVENT column lists the events for which active sessions are waiting. The "sqlnet message from client" wait event indicates that the shared or foreground process is waiting for a message from a client. If this wait event has occurred, you can check to see whether the message has been sent by the user or received by Oracle.

You can investigate hangups by looking at V$SESSION_WAIT to see what the sessions are waiting for. If a client has sent a message, you can determine whether Oracle is responding to it or is still waiting for it.

In V$SESSTAT you can see the number of bytes that have been received from the client, the number of bytes sent to the client, and the number of calls the client has made.

Solving Network Problems

This section describes several techniques for enhancing performance and solving network problems.

- Using Array Interfaces
- Using Prestarted Processes
- Adjusting Session Data Unit Buffer Size
- Increasing the Listener Queue Size
- Using TCP.NODELAY
- Using Shared Server Processes Rather than Dedicated Server Processes
- Using Connection Manager

See Also: The *Net8 Administrator's Guide.*

Using Array Interfaces

Reduce network calls by using array interfaces. Instead of fetching one row at a time, it is more efficient to fetch ten rows with a single network round trip.

See Also: *Pro*C/C++ Precompiler Programmer's Guide* and *Pro*COBOL Precompiler Programmer's Guide* for more information on array interfaces.

Using Prestarted Processes

Prestarting processes can improve connect time with a dedicated server. This is particularly true of heavily loaded systems not using multi-threaded servers, where connect time is slow. If prestarted processes are enabled, the listener can hand off the connection to an existing process with no wait time whenever a connection request arrives. Connection requests do not have to wait for new processes to be started.

Adjusting Session Data Unit Buffer Size

Before sending data across the network, Net8 buffers data into the Session Data Unit (SDU). It sends the data stored in this buffer when the buffer is full or when an application tries to read the data. When large amounts of data are being retrieved and when packet size is consistently the same, it may speed retrieval to adjust the default SDU size.

Optimal SDU size depends on the normal packet size. Use a sniffer to find out the frame size, or set tracing on to its highest level to check the number of packets sent and received and to determine whether they are fragmented. Tune your system to limit the amount of fragmentation.

Use Oracle Network Manager to configure a change to the default SDU size on both the client and the server; SDU size should generally be the same on both.

See Also: The *Net8 Administrator's Guide.*

Increasing the Listener Queue Size

The network listener active on the database server monitors and responds to connection requests. You can increase the listening queue for a listening process in order to handle larger numbers of concurrent requests dynamically.

Using TCP.NODELAY

When a session is established, Net8 packages and sends data between server and client using packets. Use the TCP.NODELAY option, which causes packets to be flushed on to the network more frequently. If you are streaming large amounts of data, there is no buffering and hence no delay.

Although Net8 supports many networking protocols, TCP tends to have the best scalability.

See Also: Your platform-specific Oracle documentation.

Using Shared Server Processes Rather than Dedicated Server Processes

Shared server processes, such as multi-threaded server dispatchers, tend to provide better performance than dedicated server processes. Dedicated server processes are committed to one session only and exist for the duration of that session. In contrast, a shared server process enables many clients to connect to the same server without the need for a dedicated server process for each client. A dispatcher handles multiple incoming session requests to the shared server.

Note: The Oracle default is dedicated server processing. For more information on the multi-threaded server, please refer to *Oracle8i Concepts* and to the *Oracle8i Administrator's Guide*.

Using Connection Manager

In Net8 you can use the Connection Manager to conserve system resources by multiplexing: funneling many client sessions through a single transport connection to a server destination. In this way you can increase the number of sessions that a process can handle.

Use Connection Manager to control client access to dedicated servers. In addition, Connection Manager provides multiple protocol support allowing a client and server with different networking protocols to communicate.

See Also: The *Net8 Administrator's Guide*.

23 Tuning the Multi-Threaded Server Architecture

The Multi-threaded Server (MTS) is a strategic component of Oracle server technology that provides greater user scalability for applications supporting numerous clients with concurrent database connections. Applications benefit from MTS features such as connection pooling and multiplexing.

MTS offers strategic functionality for Oracle because its architecture allows increased user scalability, improved throughput, and better response time. MTS does this while using fewer resources, even as the number of users increases. MTS can scale most applications to accommodate a large number of users without changing the applications.

Note: You can also configure Oracle to use both MTS and dedicated server configurations. For more information on this, please refer to the *Net8 Administrator's Guide.*

Setting Up MTS

To set up MTS, set the MTS-related parameters to suit your application.

See Also: For details on how to do this, please refer to the *Oracle8i Administrator's Guide*

Application Types that Benefit from MTS

Applications requiring a large number of concurrent database user connections while minimizing database server memory use are the types of applications that benefit most from MTS. Examples of these are OLTP applications with high transaction volumes and web-based, thin-client applications using IIOP with Java-based architectures.

MTS provides user scalability and performance enhancements to enable using a three-tier application model with the Oracle Server. In many cases, MTS is an excellent replacement for transaction processing (TP) monitors. This is because MTS does not have some of the performance overhead associated with TP monitors.

You can also have an excellent scalability, high-availability system by using MTS along with OPS (Oracle Parallel Server). It is possible to achieve 24-hours per day, seven days-per-week uptime when using MTS with OPS.

The following are among the key performance enhancements and new MTS-related functionality for Oracle8*i*:

- Asynchronous network IO
- Major memory management enhancements
- Support for dynamic multiple presentations
- Improved manageability and significantly easier configuration
- Connection pooling and connection multiplexing
- Improved load balancing algorithm; this feature is also known as "node load balancing"

See Also: For more information on OPS please refer to *Oracle8i Parallel Server Concepts and Administration*.

Improving User Scalability with MTS

Using MTS allows you to tune your system and minimize resource usage. The values for MTS-related parameters for typical applications are determined by a number of factors such as:

- Ratios of connections to dispatchers
- Connections to shared servers
- The application itself

The following section describes how to tune dispatchers and the use of MTS' connection pooling and connection multiplexing features.

Configuring Dispatchers

The number of active dispatchers on a running system does not increase dynamically. To increase the number of dispatchers to accommodate more users after system startup, alter the value for the DISPATCHERS attribute of the MTS_DISPATCHERS parameter. You can use a value for this parameter such that the total number of dispatchers requested does not exceed the value set for the parameter MTS_MAX_DISPATCHERS. The default value of MTS_MAX_DISPATCHERS is 5.

If the number of dispatchers you configure is greater than the value of MTS_MAX_DISPATCHERS, Oracle increases the value of MTS_MAX_DISPATCHERS to this value. Unless you expect the number of concurrent connections to increase over time, you do not need to set this parameter.

A ratio of 1 dispatcher for every 250 connections works well for typical systems. For example, if you anticipate 1,000 connections at peak time, you may want to configure 4 dispatchers. Being too aggressive in your estimates is not beneficial; configuring too many dispatchers can degrade performance.

Note: On NT, dispatchers are threads and not separate processes.

Connection Pooling and Connection Multiplexing

If you do not have the resources to configure a large number of dispatchers and your system requires more simultaneous connections, use connection pooling and connection multiplexing with MTS.

MTS enables connection pooling where clients share connection slots in a pool on the server side by temporarily releasing client connections when connections are idle. MTS enables connection multiplexing with the connection manager. This allows multiple client connections to share a single connection from the connection manager to the database.

Note: The connection pooling and multiplexing features only work with MTS.

See Also: For an example of implementing multiplexing, please refer to the *Net8 Administrator's Guide.*

Maximizing Throughput and Response Time with MTS

This section explains how to maximize throughput and response time using MTS by covering the following topics:

- Configuring and Managing the Number of Shared Servers
- Tuning the SDU Size

Configuring and Managing the Number of Shared Servers

The number of shared server processes spawned changes dynamically based on need. The system begins by spawning the number of shared servers as initially designated by the value set for MTS_SERVERS. If needed, the system spawns more shared servers up to the value set for the MTS_MAX_SERVERS parameter.

If system load decreases, it maintains a minimum number of servers as specified by MTS_SERVERS. Because of this, do not set MTS_SERVERS to too high a value at system startup. Typical systems seem to stabilize at a ratio of 1 shared server for every 10 connections.

For OLTP applications, the connections-to-servers ratio could be higher since the rate of requests could be low and the ratio of server usage-to-requests is low. In applications where the rate of requests is high or the server usage-to-request ratio is high, the connections-to-servers ratio could be lower.

In this case, set MTS_MAX_SERVERS to a reasonable value based on your application. The default value of MTS_MAX_SERVERS is 20 and the default for MTS_SERVERS is 1.

On NT, exercise care when setting MTS_MAX_SERVERS to too a high value because as mentioned, each server is a thread in a common process. The optimal values for these settings can change based on your configuration; these are just estimates of what seems to work for typical configurations.

MTS_MAX_SERVERS is a static INIT.ORA parameter, so you cannot change it without shutting down your database. However, MTS_SERVERS is a dynamic parameter, so you can change it within an active sessions using the ALTER SYSTEM command.

Example OLTP Application: If you expect to require 2,000 concurrent connections, begin with 200 shared servers or 1 shared server for every 10 connections. Set MTS_MAX_SERVERS to 400. Since this is an OLTP application, you would expect the load imposed by each connection on the shared servers to be lower than a typical application. Instead, start off with 100 shared servers, not 200. If you need more shared servers, the system adjusts the number up to the MTS_MAX_SERVERS value.

Tuning the SDU Size

Net8 stores buffer data in the Session Data Unit (SDU) and sends data stored in this buffer when it is full or when applications try to read the data. Tune the default SDU size when large amounts of data are being retrieved and when packet size is consistently the same. This may speed retrieval and also reduce fragmentation.

See Also: For more information on this, please refer to "Adjusting Session Data Unit Buffer Size" on page 22-3.

Balancing Load Connections

Connection load balancing distributes the load based on:

- The number of connections per dispatcher, also known as "dispatcher level load balancing"
- The load on the node, also known as "node level load balancing"

Connection load balancing only works when MTS is enabled.

When using OPS or replicated databases, the connection load balancing feature of MTS provides better load balancing than if you use DESCRIPTION_LISTS. This is because this feature distributes client connections based on actual CPU load. MTS also enables simplified application-dependent routing configurations that ensure that a request from the same application may be routed to the same node each time. This improves the efficiency of application data transfer.

Note: Connection load balancing only works with MTS enabled.

See Also: For more information, also refer to the chapter "Administering Multiple Instances" in *Oracle8i Parallel Server Concepts and Administration*.

Tuning Memory Use with MTS

This section explains how to tune memory use with MTS.

Configuring the Large Pool and Shared Pool for MTS

Oracle recommends using the large pool to allocate MTS-related UGA (User Global Area), not the shared pool. This is because Oracle also uses the shared pool to allocate SGA memory for other purposes such as shared SQL and PL/SQL procedures. Using the large pool instead of the shared pool also decreases SGA fragmentation.

To store MTS-related UGA in the large pool, specify a value for the parameter LARGE_POOL_SIZE. If you do not set a value for LARGE_POOL_SIZE, Oracle uses the shared pool for MTS user session memory.

When using MTS, configure a larger-than-normal large or shared pool. This is necessary because MTS stores all user state information from the UGA in the SGA (Shared Global Area).

If you use the shared pool, Oracle has a default value for SHARED_POOL_SIZE of 8MB on 32-bit systems and 64MB on 64 bit systems. The LARGE_POOL_SIZE does not have a default value, but its minimal value is 300K.

Note: Oracle still allocates some fixed amount of memory per session from the shared pool even if you set a value for LARGE_POOL_SIZE.

Determining an Effective Setting for MTS UGA Storage

The exact amount of UGA Oracle uses depends on each application. To determine an effective setting for the large or shared pools, observe UGA use for a typical user and multiply this amount by the estimated number of user sessions.

Even though use of shared memory increases with MTS, the total amount of memory use decreases. This is because there are fewer processes, therefore, Oracle uses less PGA memory with MTS. This is the opposite of how this functions in dedicated server environments.

Limiting Memory Use Per User Session by Setting PRIVATE_SGA

With MTS, you can set the PRIVATE_SGA parameter to limit the memory used by each client session from the SGA. PRIVATE_SGA defines the number of bytes of memory used from the SGA by a session. However, this parameter is rarely used because most DBAs do not limit SGA consumption an a user-by-user basis.

See Also: For more information, please refer to the *Oracle8i Reference.*

MTS-related Views with Connection, Load and Statistics Data

Oracle provides several views with information about dispatchers, shared servers, the rate at which connections are established, the messages queued, shared memory used, and so on.

V$DISPATCHER_RATE	This view gives information and statistics about the rate at which each dispatcher is receiving and handling messages, events, and so on.
V$MTS	This view gives statistics about things like the maximum number of shared servers ever started by the system, maximum number of connections handled by a dispatcher, and so on, since the instance was started.
V$DISPATCHER	This view gives information about dispatcher processes.
V$SHARED_SERVERS	This view gives information about shared server processes.
V$CIRCUITS	This view gives information about the virtual circuits. Virtual circuits are state objects that act as repositories of all the user related state information that needs to be accessed during a database session. There is one virtual circuit per client connection.
V$QUEUE	This view gives information about messages in the common message queue and the dispatcher message queues.
V$SGA	This view gives information about the system global area.
V$SGASTAT	This view gives detailed statistical information about the system global area.
V$SHARED_POOL_RESERVED	This view gives statistics.

MTS Feature Performance Issues

Performance of certain database features may degrade slightly when MTS is used. These features include Bfiles, parallel execution, inter-node parallel execution, and hash joins. This is because these features may prevent a session from migrating to another shared server while they are active.

A session may remain non-migratable after a request from the client has been processed. Use of these features may make sessions non migratable because these features have not stored all the user state information in the UGA, but have left some of the state in the PGA. As a result, if different shared servers process requests from the client, the part of the user state stored in the PGA is inaccessible. To avoid this, individual shared servers often need to remain bound to a user session. This makes the session non-migratable among shared servers.

When using these features, you may need to configure more shared servers. This is because some servers may be bound to sessions for an excessive amount of time.

24

Tuning the Operating System

This chapter explains how to tune the operating system for optimal performance of the Oracle server. Topics include:

- Understanding Operating System Performance Issues
- Detecting Operating System Problems
- Solving Operating System Problems

See Also: In addition to information in this chapter, please refer to your operating system specific documentation.

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 better performance, verify your work or try to reduce system time. Make sure there is enough I/O bandwidth, CPU power, and swap space. Do not expect, however, that further tuning of the operating system will have a significant effect on application performance. Changes in the Oracle configuration or in the application are likely to make a more significant difference in operating system efficiency than simply tuning the operating system.

For example, if your application experiences excessive buffer busy waits, the number of system calls will increase. If you reduce the buffer busy waits by tuning the application, then the number of system calls will decrease. Similarly, if you turn on the Oracle initialization parameter TIMED_STATISTICS, then the number of system calls will increase. If you turn it off, then system calls will decrease.

See Also: For detailed information, see your Oracle platform-specific documentation and your operating system vendor's documentation.

Operating System and Hardware Caches

Operating systems and device controllers provide data caches that do not directly conflict with Oracle's own cache management. Nonetheless, these structures can consume resources while offering little or no benefit to performance. This is most noticeable on a UNIX system that has the database container 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 container 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.

On NT this problem does not arise. All file requests by the database bypass the caches in the file system.

Raw Devices

Evaluate the use of raw devices on your system. Using raw devices may involve a significant amount of work, but may also provide significant performance benefits.

Raw devices impose a penalty on full table scans, but may be essential on UNIX systems if the implementation does not support "write through" cache. The UNIX file system accelerates full table scans by reading ahead when the server starts requesting contiguous data blocks. It also caches full table scans. If your UNIX system does not support the write through option on writes to the file system, it is essential that you use raw devices to ensure that at commit and checkpoint, the data that the server assumes is safely established on disk is actually there. If this is not the case, recovery from a UNIX operating system crash may not be possible.

Raw devices on NT are similar to UNIX raw devices; however, all NT devices support write through cache.

Process Schedulers

Many processes, or "threads" on NT systems, are involved in the operation of Oracle. They all access the shared memory resources in the SGA.

Be sure all Oracle processes, both background and user processes, have the same process priority. When you install Oracle, all background processes are given the default priority for your operating system. Do not change the priorities of background processes. Verify that all user processes have the default operating system priority.

Assigning different priorities to Oracle processes may exacerbate the effects of contention. Your operating system may not grant processing time to a low-priority process if a high-priority process also requests processing time. If a high-priority process needs access to a memory resource held by a low-priority process, the high-priority process may wait indefinitely for the low-priority process to obtain the CPU, process the request, and release the resource.

Detecting Operating System Problems

The key statistics to extract from any operating system monitor are:

- CPU load
- Device queues
- Network activity (queues)
- Memory management (paging/swapping)

Examine CPU use to determine the ratio between the time spent running in application mode and the time spent running in operating system mode. Look at run queues to see how many processes are runable and how many system calls are being executed. See if paging or swapping is occurring, and check the number of I/Os being performed.

> **See Also:** Your Oracle platform-specific documentation and your operating system vendor's documentation.

Solving Operating System Problems

This section provides hints for tuning various systems by explaining the following topics:

- Performance on UNIX-Based Systems
- Performance on NT Systems
- Performance on Mainframe Computers

Familiarize yourself with platform-specific issues so you know what performance options your operating system provides. For example, some platforms have post wait drivers that allow you to map system time and thus reduce system calls, enabling faster I/O.

> **See Also:** Your Oracle platform-specific documentation and your operating system vendor's documentation.

Performance on UNIX-Based Systems

On UNIX systems, try to establish a good ratio between the amount of time the operating system spends fulfilling system calls and doing process scheduling, and the amount of time the application runs. Your goal should be running 60% to 75% of the time in application mode, and 25% to 40% of the time in operating system mode. If you find that the system is spending 50% of its time in each mode, then determine what is wrong.

The ratio of time spent in each mode is only a symptom of the underlying problem, which might involve:

- Swapping
- Executing too many O/S system calls
- Running too many processes

If such conditions exist, there is less time available for the application to run. The more time you can release from the operating system side, the more transactions your application can perform.

Performance on NT Systems

On NT systems, as with UNIX-based systems, you should establish an appropriate ratio between time in application mode and time in system mode. On NT you can easily monitor many factors with Performance Monitor: CPU, network, I/O, and memory are all displayed on the same graph, to assist you in avoiding bottlenecks in any of these areas.

Performance on Mainframe Computers

Consider the paging parameters on a mainframe, and remember that Oracle can exploit a very large working set of parameters.

Free memory in VAX/VMS environments is actually memory that is not mapped to any operating system process. On a busy system, free memory likely contains a page belonging to one or more currently active process. When that access occurs, a "soft page fault" takes place, and the page is included in the working set for the process. If the process cannot expand its working set, then one of the pages currently mapped by the process must be moved to the free set.

Any number of processes may have pages of shared memory within their working sets. The sum of the sizes of the working sets can thus markedly exceed the available memory. When the Oracle server is running, the SGA, the Oracle kernel code, and the Oracle Forms runtime executable are normally all sharable and account for perhaps 80% or 90% of the pages accessed.

Adding more buffers is not necessarily better. Each application has a threshold number of buffers at which the cache hit ratio stops rising. This is typically quite low (approximately 1500 buffers). Setting higher values simply increases the management load for both Oracle and the operating system.

25

Tuning Instance Recovery Performance

This chapter offers guidelines for tuning instance recovery. It includes the following topics:

Understanding Instance Recovery

Instance and crash recovery are the automatic application of redo log records to Oracle data blocks after a crash or system failure. If either a single instance database crashes or all instances of an OPS (Oracle Parallel Server) configuration crash, Oracle performs instance recovery at the next startup. If one or more instances of an OPS configuration crash, a surviving instance performs recovery.

Instance and crash recovery occur in two phases. In phase one, Oracle applies all committed and uncommitted changes in the redo log files to the affected datablocks. In phase two, Oracle applies information in the rollback segments to undo changes made by uncommitted transactions to the data blocks.

How Oracle Applies Redo Log Information

During normal operations, Oracle's DBW*n* processes periodically write dirty buffers, or buffers that have in-memory changes, to disk. Periodically, Oracle records the highest SCN of all changes to blocks such that all data blocks with changes below that SCN have been written to disk by DBW*n*. This SCN is the "checkpoint".

Records that Oracle appends to the redo log file after the change record that the checkpoint refers to are changes that Oracle has not yet written to disk. If a failure occurs, only redo log records containing changes at SCNs higher than the checkpoint need to be replayed during recovery.

The duration of recovery processing is directly influenced by the number of data blocks that have changes at SCNs higher than the SCN of the checkpoint. For example, Oracle will recover a redo log with 100 entries affecting one data block more quickly than it recovers a redo log with 10 entries for 10 different data blocks. This is because for each log record processed during recovery, the corresponding data block must be read from disk by Oracle so that the change represented by the redo log entry can be applied to that block.

Trade-offs of Minimizing Recovery Duration

The principal means of balancing the duration of instance recovery and daily performance is by influencing how aggressively Oracle advances the checkpoint. If you force Oracle to keep the checkpoint only a few blocks behind the most recent redo log record, you minimize the number of blocks Oracle processes during recovery.

The trade-off for having minimal recovery time, however, is increased performance overhead for normal database operations. If daily operational efficiency is more important than minimizing recovery time, decreasing the frequency of writes to the datafiles increases instance recovery time.

Tuning the Duration of Instance and Crash Recovery

There are several methods for tuning instance and crash recovery to keep the duration of recovery within user-specified bounds. These methods are:

- Using initialization parameters to influence the number of redo log records and data blocks involved in recovery
- Sizing the redo log file to influence checkpointing frequency
- Using SQL statements to initiate checkpoints
- Parallelizing instance recovery operations to further shorten the recovery duration

The Oracle8*i* Enterprise Edition also offers Fast-start fault recovery functionality to control instance recovery.

Using Initialization Parameters to Influence Instance and Crash Recovery Time

During recovery, Oracle performs two main tasks:

- Read the redo logs to determine what has been changed
- Read data blocks to determine whether to apply changes

You can use three initialization parameters to influence how aggressively Oracle advances the checkpoint as shown in Table 25–1:

Table 25–1 Initialization Parameters Influencing Checkpoints

Parameter	Purpose
LOG_CHECKPOINT_TIMEOUT	Limit the number of seconds between the most recent redo record and the checkpoint.
LOG_CHECKPOINT_INTERVAL	Limit the number of redo records between the most recent redo record and the checkpoint.
FAST_START_IO_TARGET	Limit instance recovery time by controlling the number of data blocks Oracle processes during instance recovery.

Note: You can only use the FAST_START_IO_TARGET parameter with the Oracle8*i* Enterprise Edition.

Using LOG_CHECKPOINT_TIMEOUT to Influence Recovery

Set the initialization parameter LOG_CHECKPOINT_TIMEOUT to a value *n* (where *n* is an integer) to require that the latest checkpoint position follow the most recent redo block by no more than *n* seconds. In other words, at most, *n* seconds' worth of logging activity can occur between the most recent checkpoint position and the end of the redo log. This forces the checkpoint position to keep pace with the most recent redo block

You can also interpret LOG_CHECKPOINT_TIMEOUT as specifying an upper bound on the time a buffer can be dirty in the cache before DBW*n* must write it to disk. For example, if you set LOG_CHECKPOINT_TIMEOUT to 60, then no buffers remain dirty in the cache for more than 60 seconds. The default value for LOG_CHECKPOINT_TIMEOUT is 1800.

Note: The minimum value for LOG_CHECKPOINT_TIMEOUT in the Standard Edition is 900. If you set the value below 900 in the Standard Edition, Oracle rounds it to 900.

Using LOG_CHECKPOINT_INTERVAL to Influence Recovery

Set the initialization parameter LOG_CHECKPOINT_INTERVAL to a value *n* (where *n* is an integer) to require that the checkpoint position never follow the most recent redo block by more than *n* blocks. In other words, at most *n* redo blocks can exist between the checkpoint position and the last block written to the redo log. In effect, you are limiting the amount of redo blocks that can exist between the checkpoint and the end of the log.

Oracle limits the maximum value of LOG_CHECKPOINT_INTERVAL to 90% of the smallest log to ensure that the checkpoint advances far enough to eliminate "log wrap". Log wrap occurs when Oracle fills the last available redo log file and cannot write to any other log file because the checkpoint has not advanced far enough. By ensuring that the checkpoint never gets too far from the end of the log, Oracle never has to wait for the checkpoint to advance before it can switch logs.

LOG_CHECKPOINT_INTERVAL is specified in redo blocks. Redo blocks are the same size as operating system blocks. Use the LOG_FILE_SIZE_REDO_BLKS column in V$INSTANCE_RECOVERY to see the number of redo blocks corresponding to 90% of the size of the smallest log file.

Using FAST_START_IO_TARGET to Influence Instance Recovery Time

You can only use the initialization parameter FAST_START_IO_TARGET if you have the Oracle8*i* Enterprise Edition. You can set this parameter to *n*, where *n* is an integer limiting to *n* the number of buffers that Oracle processes during crash or instance recovery. Because the number of I/Os to be processed during recovery correlates closely to the duration of recovery, the FAST_START_IO_TARGET parameter gives you the most precise control over the duration of recovery.

FAST_START_IO_TARGET advances the checkpoint because DBW*n* uses the value of FAST_START_IO_TARGET to determine how much writing to do. Assuming that users are making many updates to the database, a low value for this parameter forces DBW*n* to write changed buffers to disk. The CKPT process reflects this progress as the checkpoint advances. Of course, if user activity is low or non-existent, DBW*n* does not have any buffers to write, so the checkpoint does not advance.

The smaller the value of FAST_START_IO_TARGET, the better the recovery performance, since fewer blocks require recovery. If you use smaller values for this parameter, however, you impose higher overhead during normal processing, since DBW*n* must write more buffers to disk more frequently.

> **See Also:** For more information, see "Estimating Recovery Time" on page 25-3 and "Calculating Performance Overhead" on page 25-11. For more information about initialization parameters, see the *Oracle8i Reference*.

Using Redo Log Size to Influence Checkpointing Frequency

The size of a redo log file directly influences checkpoint performance. The smaller the size of the smallest log, the more aggressively Oracle writes dirty buffers to disk to ensure the position of the checkpoint has advanced to the current log before that log completely fills. Oracle enforces this behavior by ensuring the number of redo blocks between the checkpoint and the most recent redo record is less than 90% of the size of the smallest log.

If your redo logs are small compared to the number of changes made against the database, Oracle must switch logs frequently. If the value of LOG_CHECKPOINT_INTERVAL is less than 90% of the size of the smallest log, this parameter will have the most influence over checkpointing behavior.

Although you specify the number and sizes of online redo log files at database creation, you can alter the characteristics of your redo log files after startup. Use the ADD LOGFILE clause of the ALTER DATABASE command to add a redo log file and specify its size, or the DROP LOGFILE clause to drop a redo log.

The size of the redo log appears in the LOG_FILE_SIZE_REDO_BLKS column of the V$INSTANCE_RECOVERY dynamic performance. This value shows how the size of the smallest online redo log is affecting checkpointing. By increasing or decreasing the size of your online redo logs, you indirectly influence the frequency of checkpoint writes.

> **See Also:** For information on using the V$INSTANCE_RECOVERY view to tune instance recovery, see "Estimating Recovery Time" on page 25-9.

Using SQL Statements to Initiate Checkpoints

Besides setting initialization parameters and sizing your redo log files, you can also influence checkpoints with SQL statements. ALTER SYSTEM CHECKPOINT directs Oracle to record a checkpoint for the node, and ALTER SYSTEM CHECKPOINT GLOBAL directs Oracle to record a checkpoint for every node in a cluster.

SQL-induced checkpoints are "heavyweight". This means Oracle records the checkpoint in a control file shared by all the redo threads. Oracle also updates the datafile headers. SQL-induced checkpoints move the checkpoint position to the point that corresponded to the end of the log when the command was initiated. These checkpoints can adversely affect performance because the additional writes to the datafiles increase system overhead.

See Also: For more information about these statements, see the *Oracle8i SQL Reference*.

Monitoring Instance Recovery

Use the V$INSTANCE_RECOVERY view to see your current recovery parameter settings. You can also use statistics from this view to calculate which parameter has the greatest influence on checkpointing. V$INSTANCE_RECOVERY contains columns as shown in Table 25–2:

Table 25–2 V$INSTANCE_RECOVERY

Column	Description
RECOVERY_ESTIMATED_IOS	The estimated number of data blocks to be processed during recovery based on the in-memory value of the fast-start checkpoint parameter.
ACTUAL_REDO_BLKS	The current number of redo blocks required for recovery.
TARGET_REDO_BLKS	The goal for the maximum number of redo blocks to be processed during recovery. This value is the minimum of the following 4 columns.
LOG_FILE_SIZE_REDO_BLKS	The number of redo blocks to be processed during recovery to guarantee that a log switch never has to wait for a checkpoint.
LOG_CHKPT_TIMEOUT_REDO_BLKS	The number of redo blocks that need to be processed during recovery to satisfy LOG_CHECKPOINT_TIMEOUT.

Table 25–2 V$INSTANCE_RECOVERY

Column	Description
LOG_CHKPT_INTERVAL_REDO_BLKS	The number of redo blocks that need to be processed during recovery to satisfy LOG_CHECKPOINT_INTERVAL.
FAST_START_IO_TARGET_REDO_BLKS	The number of redo blocks that need to be processed during recovery to satisfy FAST_START_IO_TARGET.

The value appearing in the TARGET_REDO_BLKS column equals a value appearing in another column in the view. This other column corresponds to the parameter or log file that is determining the maximum number of redo blocks that Oracle processes during recovery. The setting for the parameter in this column is imposing the heaviest requirement on redo block processing.

Determining the Strongest Checkpoint Influence: Scenario

As an example, assume your initialization parameter settings are as follows:

```
FAST_START_IO_TARGET = 1000
LOG_CHECKPOINT_TIMEOUT = 1800 # default
LOG_CHECKPOINT_INTERVAL = 0# default: disabled interval checkpointing
```

You execute the query:

```
SELECT * FROM V$INSTANCE_RECOVERY;
```

Oracle responds with:

```
RECOVERY_EST   ACTUAL_REDO_   TARGET_REDO_   LOG_FILE_SIZ   LOG_CHKPT_TI   LOG_CHKPT_IN   FAST_START_I
IMATED_IOS     BLKS           BLKS           E_REDO_ BLKS   MEOUT_REDO_B   TERVAL_REDO_   O_TARGET_BLK
                                                            LKS            BLKS           S
-------------------------------------------------------------------------------------------------------
        1025           6169           4215          55296          35485     4294967295           4215
1 row selected.
```

As you can see by the values in the last three columns, the FAST_START_IO_TARGET parameter places heavier recovery demands on Oracle than the other two parameters: it requires that Oracle process no more than 4,215 redo blocks during recovery. The LOG_FILE_SIZE_REDO_BLKS column indicates that Oracle can process up to 55,296 blocks during recovery, so the log file size is not the heaviest influence on checkpointing.

Note: The value for LOG_CHKPT_INTERVAL_REDO_BLKS, 4294967295, corresponds to the maximum possible value indicating that this column does not have the greatest influence over checkpointing.

The TARGET_REDO_BLKS column shows the smallest value of the last five columns. This shows the parameter or condition that exerts the heaviest requirement for Oracle checkpointing. In this example, the FAST_START_IO_TARGET parameter is the strongest influence with a value of 4,215.

Assume you make several updates to the database and query V$INSTANCE_RECOVERY three hours later. Oracle responds with the following:

```
 RECOVERY_    ACTUAL_      TARGET_      LOG_FILE_    LOG_CHKPT_   LOG_CHKPT_   FAST_START_
 ESTIMATED_   REDO_BLKS    REDO_BLKS    SIZE_REDO_   TIMEOUT_     INTERVAL_    IO_TARGET_
 IOS                                    BLKS         REDO_BLKS    REDO_BLKS    BLKS
-------------------------------------------------------------------------------------------
        1022          916          742        55296        44845   4294967295          742
1 row selected.
```

FAST_START_IO_TARGET is still exerting the strongest influence over checkpointing behavior, although the number of redo blocks corresponding to this target has changed dramatically. This change is not due to a change in FAST_START_IO_TARGET or the corresponding RECOVERY_ESTIMATED_IOS. Instead, this indicates that operations requiring I/O in the event of recovery are more frequent in the redo log, so fewer redo blocks now correspond to the same FAST_START_IO_TARGET.

Assume you decide that FAST_START_IO_TARGET is placing an excessive limit on the maximum number of redo blocks that Oracle processes during recovery. You adjust FAST_START_IO_TARGET to 8000, set LOG_CHECKPOINT_TIMEOUT to 60, and perform several updates. You reissue the query to V$INSTANCE_RECOVERY and Oracle responds with:

```
 RECOVERY_    ACTUAL_      TARGET_      LOG_FILE_    LOG_CHKPT_   LOG_CHKPT_   FAST_START_
 ESTIMATED_   REDO_BLKS    REDO_BLKS    SIZE_REDO_   TIMEOUT_     INTERVAL_    IO_TARGET_
 IOS                                    BLKS         REDO_BLKS    REDO_BLKS    BLKS
-------------------------------------------------------------------------------------------
        1640         6972         6707        55296         6707   4294967295        10338
1 row selected.
```

Because the TARGET_REDO_BLKS column value of 6707 corresponds to the value in the LOG_CHKPT_TIMEOUT_REDO_BLKS column, LOG_CHECKPOINT_TIMEOUT is now exerting the most influence over checkpointing behavior.

Estimating Recovery Time

Use statistics from the V$INSTANCE_RECOVERY view to estimate recovery time using the following formula:

$$\frac{\textit{RECOVERY\_ESTIMATED\_JOBS}}{\textit{Maximum I/Os per second that your system can perform}}$$

For example, if RECOVERY_ESTIMATED_IOS is 2,500, and the maximum number of writes your system performs is 500 per second, then recovery time is 5 seconds. Note the following restrictions:

- The value for the maximum I/Os per second the system can perform is difficult to measure accurately
- There is no guarantee the system will sustain the I/O rate during recovery
- This estimate for recovery time is only valid when FAST_START_IO_TARGET is both enabled and when this parameter is the determining influence on checkpointing behavior

To adjust recovery time, change the initialization parameter that has the most influence over checkpointing. Use the V$INSTANCE_RECOVERY view as described in "Monitoring Instance Recovery" on page 25-6 to determine which parameter to adjust. Then either adjust the parameter to decrease or increase recovery time as required.

Adjusting Recovery Time: Example Scenario

As an example, assume as in "Determining the Strongest Checkpoint Influence: Scenario" on page 25-7 that your initialization parameter settings are as follows:

```
FAST_START_IO_TARGET = 1000
LOG_CHECKPOINT_TIMEOUT = 1800 # default
LOG_CHECKPOINT_INTERVAL = 0 # default: disabled interval checkpointing
```

You execute the query:

```
SELECT * FROM V$INSTANCE_RECOVERY;
```

Oracle responds with:

```
  RECOVERY_       ACTUAL_         TARGET_         LOG_FILE_       LOG_CHKPT_      LOG_CHKPT_      FAST_START_
  ESTIMATED_      REDO_BLKS       REDO_BLKS       SIZE_REDO_      TIMEOUT_        INTERVAL_       IO_TARGET_
  IOS                                             BLKS            REDO_BLKS       REDO_BLKS       BLKS
-------------------------------------------------------------------------------------------------------------
          1025            6169            4215           55296           35485      4294967295            4215
1 row selected.
```

You calculate recovery time using the formula on page 25-9, where RECOVERY_ESTIMATED_JOBS is 1025 and the maximum I/Os per second the system can perform is 500:

$$\frac{1025}{500} = 2.05$$

You decide you can afford slightly more than 2.05 seconds of recovery time: constant access to the data is not critical. You increase the value for the parameter FAST_START_IO_TARGET to 2000 and perform several updates. You then reissue the query and Oracle displays:

```
  RECOVERY_       ACTUAL_         TARGET_         LOG_FILE_       LOG_CHKPT_      LOG_CHKPT_      FAST_START_
  ESTIMATED_      REDO_BLKS       REDO_BLKS       SIZE_REDO_      TIMEOUT_        INTERVAL_       IO_TARGET_
  IOS                                             BLKS            REDO_BLKS       REDO_BLKS       BLKS
-------------------------------------------------------------------------------------------------------------
  2007            8301            8012            55296           40117           4294967295      8012
1 row selected.
```

Recalculate recovery time using the same formula:

$$\frac{2007}{500} = 4.01$$

You have increased your recovery time by 1.96 seconds. If you can afford more time, repeat the procedure until you arrive at an acceptable recovery time.

Calculating Performance Overhead

To calculate performance overhead, use the V$SYSSTAT view. For example, assume you execute the query:

```
SELECT NAME, VALUE FROM V$SYSSTAT
WHERE NAME IN ( 'PHYSICAL READS', 'PHYSICAL WRITES',);
```

Oracle responds with:

```
NAME                                 VALUE
physical reads                        2376
physical writes                      14932
physical writes non checkpoint       11165
3 rows selected.
```

The first row shows the number of data blocks retrieved from disk. The second row shows the number of data blocks written to disk. The last row shows the value of the number of writes to disk that would occur if you turned off checkpointing.

Use this data to calculate the overhead imposed by setting the FAST_START_IO_TARGET initialization parameter. To effectively measure the percentage of extra writes, mark the values for these statistics at different times, T_1 and T_2. Use the following formula where the variables stand for the following:

Variable	Definition
*_1	Value of prefixed variable at time T_1, which is any time after the database has been running for a while
*_2	Value of prefixed variable at time T_2, which is later than T_1 and not immediately after changing any of the checkpoint parameters
PWNC	Physical writes non checkpoint
PW	Physical writes
PR	Physical reads
EIO	Percentage of estimated extra I/Os generated by enabling checkpointing

Calculate the percentage of extra I/Os generated by fast-start checkpointing using this formula:

$$[((PW\_2 - PW\_1) - (PWNC\_2 - PWNC\_1)) / ((PR\_2 - PR\_1) + (PW\_2 - PW\_1))] \times 100\% = EIO$$

It can take some time for database statistics to stabilize after instance startup or dynamic initialization parameter modification. After such events, wait until all blocks age out of the buffer cache at least once before taking measurements.

If the percentage of extra I/Os is too high, increase the value for FAST_START_IO_TARGET. Adjust this parameter until you get an acceptable value for the RECOVERY_ESTIMATED_IOS in V$INSTANCE_RECOVERY as described in "Determining the Strongest Checkpoint Influence: Scenario" on page 25-7.

The number of extra writes caused by setting FAST_START_IO_TARGET to a non-zero value is application-dependent. An application that repeatedly modifies the same buffers incurs a higher write penalty because of Fast-start checkpointing than an application that does not. The extra write penalty is not dependent on cache size.

Calculating Performance Overhead: Example Scenario

As an example, assume your initialization parameter settings are:

```
FAST_START_IO_TARGET = 2000
LOG_CHECKPOINT_TIMEOUT = 1800 # default
LOG_CHECKPOINT_INTERVAL = 0 # default: disabled interval checkpointing
```

After the statistics stabilize, you issue this query on V$SYSSTAT:

```
SELECT NAME, VALUE FROM V$SYSSTAT
WHERE NAME IN ('PHYSICAL READS', 'PHYSICAL WRITES',
'PHYSICAL WRITES NON CHECKPOINT');
```

Oracle responds with:

```
Name                                Value
physical reads                       2376
physical writes                     14932
physical writes non checkpoint      11165
3 rows selected.
```

After making updates for a few hours, you re-issue the query and Oracle responds with:

```
Name                                Value
physical reads                       3011
physical writes                     17467
physical writes non checkpoint      13231
3 rows selected.
```

Substitute the values from your select statements in the formula as described on page 25-11 to determine how much performance overhead you are incurring:

[((17467 - 14932) - (13231 - 11165)) / ((3011 - 2376) + (17467 - 14932))] x 100% = 14.8%

As the result indicates, enabling fast-start checkpointing generates about 15% more I/O than would be required had you not enabled fast-start checkpointing. After calculating the extra I/O, you decide you can afford more system overhead if you decrease recovery time.

To decrease recovery time, reduce the value for the parameter FAST_START_IO_TARGET to 1000. After items in the buffer cache age out, calculate V$SYSSTAT statistics across a second interval to determine the new performance overhead. Query V$SYSSTAT:

```
SELECT NAME, VALUE FROM V$SYSSTAT
WHERE NAME IN ('PHYSICAL READS', 'PHYSICAL WRITES',
'PHYSICAL WRITES NON CHECKPOINT');
```

Oracle responds with:

```
Name                               Value
physical reads                      4652
physical writes                    28864
physical writes non checkpoint     21784
3 rows selected.
```

After making updates, re-issue the query and Oracle responds with:

```
Name                               Value
physical reads                      6000
physical writes                    35394
physical writes non checkpoint     26438
3 rows selected.
```

Calculate how much performance overhead you are incurring using the values from your two SELECT statements:

[(35394 - 28864) - (26438 - 21784)) / ((6000 - 4652) + (35394 - 28864))] x 100% = 23.8%

After changing the parameter, the percentage of I/Os performed by Oracle is now about 24% more than it would be if you disabled Fast-start checkpointing.

Tuning the Phases of Instance Recovery

Besides using checkpoints to tune instance recovery, you can also use a variety of parameters to control Oracle's behavior during the rolling forward and rolling back phases of instance recovery. In some cases, you can parallelize operations and thereby increase recovery efficiency.

This section contains the following topics:

- Tuning the Rolling Forward Phase
- Tuning the Rolling Back Phase

Tuning the Rolling Forward Phase

Use parallel block recovery to tune the roll forward phase of recovery. Parallel block recovery uses a "division of labor" approach to allocate different processes to different data blocks during the roll forward phase of recovery. For example, if the redo log contains a substantial number of entries, process 1 takes responsibility for one part of the log file, process 2 takes responsibility for another part, process 3 takes responsibility for a third part, and so on. Crash, instance, and media recovery of many datafiles on different disk drives are good candidates for parallel block recovery.

Use the RECOVERY_PARALLELISM initialization parameter to specify the number of concurrent recovery processes for instance or media recovery operations. Because crash recovery occurs at instance startup, this parameter is useful for specifying the number of processes to use for crash recovery. The value of this parameter is also the default number of processes used for media recovery if you do not specify the PARALLEL clause of the RECOVER command. The value of this parameter must be greater than 1 and cannot exceed the value of the PARALLEL_MAX_SERVERS parameter. Parallel block recovery requires a minimum of eight recovery processes for it to be more effective than serial recovery.

Recovery is usually I/O bound on reads to data blocks. Consequently, parallelism at the block level may only help recovery performance if it increases total I/Os. In other words, parallelism at the block level by-passes operating system restrictions on asynchronous I/Os. Performance on systems with efficient asynchronous I/O typically does not improve significantly with parallel block recovery.

Tuning the Rolling Back Phase

During the second phase of instance recovery, Oracle rolls back uncommitted transactions. Oracle uses two features, Fast-start on-demand rollback and Fast-start parallel rollback, to increase the efficiency of this recovery phase.

Note: These features are part of Fast-start fault recovery and are only available in the Oracle8*i* Enterprise Edition.

This section contains the following topics:

- Using Fast-start On-demand Rollback
- Using Fast-start Parallel Rollback

Using Fast-start On-demand Rollback

Using the Fast-start on-demand rollback feature, Oracle automatically allows new transactions to begin immediately after the roll forward phase of recovery completes. Should a user attempt to access a row that is locked by a dead transaction, Oracle rolls back only those changes necessary to complete the transaction, in other words, it rolls them back "on demand." Consequently, new transactions do not have to wait until all parts of a long transaction are rolled back.

Note: Oracle does this automatically. You do not need to set any parameters or issue commands to use this feature.

Using Fast-start Parallel Rollback

In Fast-start parallel rollback, the background process SMON acts as a coordinator and rolls back a set of transactions in parallel using multiple server processes. Essentially, Fast-start parallel rollback is to rolling back what parallel block recovery is to rolling forward.

Fast-start parallel rollback is mainly useful when a system has transactions that run a long time before committing, especially parallel INSERT, UPDATE, and DELETE operations. When SMON discovers that the amount of recovery work is above a certain threshold, it automatically begins parallel rollback by dispersing the work among several parallel processes: process 1 rolls back one transaction, process 2 rolls back a second transaction, and so on. The threshold is the point at which parallel recovery becomes cost-effective, in other words, when parallel recovery takes less time than serial recovery.

One special form of Fast-start parallel rollback is intra-transaction recovery. In intra-transaction recovery, a single transaction is divided among several processes. For example, assume 8 transactions require recovery with one parallel process assigned to each transaction. The transactions are all similar in size except for transaction 5, which is quite large. This means it takes longer for one process to roll this transaction back than for the other processes to roll back their transactions.

In this situation, Oracle automatically begins intra-transaction recovery by dispersing transaction 5 among the processes: process 1 takes one part, process 2 takes another part, and so on.

You control the number of processes involved in transaction recovery by setting the parameter FAST_START_PARALLEL_ROLLBACK to one of three values:

FALSE	Turns off Fast-start parallel rollback.
LOW	Specifies that the number of recovery servers may not exceed twice the value of the CPU_COUNT parameter.
HIGH	Specifies that the number of recovery servers may not exceed four times the value of the CPU_COUNT parameter.

Parallel Rollback in an OPS Configuration In OPS, you can perform Fast-start parallel rollback on each instance. Within each instance, you can perform parallel rollback on transactions that are:

- Online on a given instance
- Offline and not being recovered on instances other than the given instance

Once a rollback segment is online for a given instance, only this instance can perform parallel rollback on transactions on that segment.

Monitoring Progress of Fast-start Parallel Rollback Monitor the progress of Fast-start parallel rollback by examining the V$FAST_START_SERVERS and V$FAST_START_TRANSACTIONS tables. V$FAST_START_SERVERS provides information about all recovery processes performing fast-start parallel rollback. V$FAST_START_TRANSACTIONS contains data about the progress of the transactions.

See Also: For more information on Fast-start parallel rollback in an OPS environment, see *Oracle8i Parallel Server Concepts and Administration*. For more information about initialization parameters, see the *Oracle8i Reference*.

Transparent Application Failover

This section covers the following topics:

- What Is Transparent Application Failover?
- How does Transparent Application Failover Work?
- Transparent Application Failover Topics for the DBA
- Transparent Application Failover Topics for Application Developers
- Transparent Application Failover Restrictions

Note: To use transparent application failover, you must have the Oracle8*i* Enterprise Edition which is described in the text, *Getting to Know Oracle8i*.

What Is Transparent Application Failover?

Transparent application failover (TAF) is the ability of applications to automatically reconnect to the database if the connection fails. If the client is not involved in a database transaction, then users may not notice the failure of the server. Because this reconnect happens automatically from within the OCI library, the client application code may not need changes to use TAF.

How does Transparent Application Failover Work?

During normal client-server database operations, the client maintains a connection to the database so the client and server can communicate. If the server fails, the connection also fails. The next time the client tries to use the connection to execute a new SQL statement, for example, the operating system displays an error to the client. Oracle most commonly then issues the error “ORA-3113: end-of-file on communication channel". At this point, the user must log in to the database again.

With TAF, however, Oracle automatically obtains a new connection to the database. This allows the user to continue to work using the new connection as if the original connection had never failed.

There several elements associated with active database connections. These can include:

- Client-Server Database Connections
- Users' Database Sessions
- Executing Commands
- Open Cursors Used for Fetching
- Active Transactions
- Server-Side Program Variables

TAF automatically restores some of these elements. Other elements, however, may need to be embedded in the application code to enable TAF to recover the connection.

Client-Server Database Connections

TAF automatically reestablishes the database connection. By default, TAF uses the same connect string to attempt to obtain a new database connection. Alternately, you can configure failover to use a different connect string; you can even pre-establish an alternate failover connection. For more information about these configurations, see "Configuring Application Failover" on page 25-21.

Users' Database Sessions

TAF automatically logs a user in with the same user ID as was used prior to failure. If multiple users were using the connection, then TAF automatically logs them in as they attempt to process database commands. Unfortunately, TAF cannot automatically restore other session properties. If the application issued ALTER SESSION commands, then the application must re-issue them after TAF processing is complete. This can be done in failover callback processing, which is described in more detail in the *Oracle Call Interface Programmer's Guide.*

Executing Commands

The client usually discovers a connection failure after a command is issued to the server that results in an error. The client cannot determine whether the command was completely executed prior to the server's failure. If the command was completely executed and it changed the state of the database, the command is not resent. If TAF reconnects in response to a command that may have changed the database, TAF issues an "ORA-25408: can not safely replay call" message to the application.

TAF automatically resends SELECT and fetch commands to the database after failover because these types of commands do not change the database's state.

Open Cursors Used for Fetching

TAF allows applications that began fetching rows from a cursor before failover to continue fetching rows after failover. This is called "select" failover. It is accomplished by re-executing a SELECT statement using the same snapshot and retrieving the same number of rows.

TAF also provides a safeguard to guarantee that the results of the select are consistent. If this safeguard fails, the application may receive the error message "ORA-25401 can not continue fetches".

Active Transactions

Any active transactions are rolled back at the time of failure because TAF cannot preserve active transactions after failover. The application instead receives the error message "ORA-25402 transaction must roll back" until a ROLLBACK is submitted.

Server-Side Program Variables

Server-side program variables, such as PL/SQL package states, are lost during failures; TAF cannot recover them. They can be initialized by making a call from the failover callback, which is described in more detail in the *Oracle Call Interface Programmer's Guide.*

Transparent Application Failover Implementation Scenarios

For TAF to effectively mask a database failure, there must be a location to which the client can reconnect. This section discusses the following database configurations, and how they work with TAF.

- OPS
- Fail Safe Systems
- Replicated Systems
- Standby Databases
- Single Instance Oracle Database

OPS

TAF was initially conceived for Oracle Parallel Server environments. All TAF functionality works with OPS, and no special setup is required. For more information about OPS, see the *Oracle8i Parallel Server Setup and Configuration Guide*.

Fail Safe Systems

You can use TAF with Oracle Fail Safe. However, since the backup instance is not available to take connections, when the primary database fails, some clients may attempt to reconnect during the time when the database server is unavailable. The failover callback may be used to get around this. For more information about failover callback, see the *Oracle Call Interface Programmer's Guide*.

Replicated Systems

TAF works with replicated systems provided that all database objects are the same on both sides of the replication. This includes the same passwords, and so on. If the data in the tables are slightly out of sync with each other, then there is a higher probability of encountering an "ORA-25401: can not continue fetches". For more information about replication, see *Oracle8i Replication*.

Standby Databases

TAF works with standby databases in a manner similar to TAF with Fail Safe. Since there may be a timeframe when a database is not available for the client to log into, the failover callback should be provided. Also, since changes made later than the most recent archive logs will not be present, there may be some data skew and hence a higher chance of encountering an "ORA-25401: can not continue fetches".

Single Instance Oracle Database

You can also use TAF in single instance Oracle database environments. After a failure in single instance environments, there can be a time period when the database is unavailable and TAF cannot re-establish a connection. For this reason, a failover callback can be used to periodically re-attempt failover. TAF successfully re-establishes the connection after the database is available again.

Transparent Application Failover Topics for the DBA

This section explains the following topics:

- Configuring TAF
- How to tell whether a client is using application failover

- Load balancing
- Planned shutdown
- Tuning TAF

Configuring Application Failover

You can configure the connect string for the application at the names server, or put it in the TNSNAMES.ORA file. Alternatively, the connect string can be hard-coded in the application.

For each application, the names server provides information about the listener, the instance group, and the failover mode. The connect string *failover_mode* field specifies the type and method of failover. For more information on syntax, please refer to the *Net8 Administrator's Guide*.

TYPE: Failover Mode Functionality Options The client's failover functionality is determined by the TYPE keyword in the connect string. The choices for the TYPE keyword are:

SELECT	This allows users with open cursors to continue fetching on them after failure. However, this mode incurs overhead on the client side in normal select operations, so the user is allowed to disable select failover.
SESSION	This fails over the session, that is, if a user's connection is lost, a second session is automatically created for the user on the backup. This type of failover does not attempt to recover selects.
NONE	This is the default, in which no failover functionality is used. This can also be explicitly specified to prevent failover from happening.

METHOD: Failover Mode Performance Options Improving the speed of application failover often requires putting more work on the backup instance. The DBA can use the METHOD keyword in the connect string to configure the BASIC or PRECONNECT performance options.

BASIC	Establish connections at failover time. This option requires almost no work on the backup server until failover time.

PRECONNECT	Pre-establish connections. This provides faster failover but requires that the backup instance be able to support all connections from every supported instance.

BACKUP: Alternate Backup Connect String In many cases it is not convenient to use the same connect string for both the initial and backup connections. In these instances, you can use the BACKUP keyword in the connect string that specifies a different TNS alias or explicit connect string for backup connections.

Failover Fields in V$SESSION

The view V$SESSION has the following fields related to failover:

FAILED_OVER	TRUE if using the backup, otherwise FALSE.
TYPE	One of SELECT, SESSION, or NONE.
METHOD	Either BASIC or PRECONNECT.

Shutting Down an Instance after Current Transactions

The TRANSACTIONAL option to the SHUTDOWN command enables you to do a planned shutdown of one instance while minimally interrupting clients. This option waits for ongoing transactions to complete. The TRANSACTIONAL option is useful for installing patch releases. Also use this option when you must bring down the instance without interrupting service.

While waiting, clients cannot start new transactions on the instance. Clients are disconnected if they attempt to start a transaction and this triggers failover if failover is enabled. When the last transaction completes, the primary instance performs a SHUTDOWN IMMEDIATE.

Disconnecting a Session after the Current Transaction

The ALTER SYSTEM DISCONNECT SESSION POST_TRANSACTION statement disconnects a session on the first call after its current transaction has been finished. The application fails over automatically.

This statement works well with TAF as a way for you to control load. If one instance is overloaded, you can manually disconnect a group of sessions using this option. Since the option guarantees there is no transaction at the time the session is disconnected, the user should never notice the change, except for a slight delay in executing the next command following the disconnect. For complete syntax of this, see the *Oracle8i SQL Reference.*

Tuning Failover Performance

The elapsed time of failover includes instance recovery as well as time needed to reconnect to the database. For best failover performance, tune instance recovery by having frequent checkpoints.

Performance can also be improved by using multiple listeners or by using the Multi-threaded Server (MTS). MTS connections tend to be much faster than connections by way of dedicated servers.

Transparent Application Failover Topics for Application Developers

This section describes multiple user handles and failover callbacks.

Multiple User Handles

Failover is supported for multiple user handles. In OCI, server context handles and user handles are decoupled. You can have multiple user handles related to the server context handle, and multiple users can thus share the same connection to the database.

If the connection is destroyed, then every user associated with that connection is failed over. But if a single user session is destroyed, then failover does not occur because the connection is still there. Failover does not reauthenticate migrateable user handles.

See Also: The *Oracle Call Interface Programmer's Guide.*

Failover Callback

Frequently failure of one instance and failover to another takes time. Because of this delay, you may want to inform users that failover is in progress. Additionally, the session on the initial instance may have received some ALTER SESSION commands. These will not be automatically replayed on the second instance. You may want to ensure that these commands are replayed on the second instance. To address such problems, you can register a callback function.

Failover calls the callback function several times when re-establishing user sessions. The first call occurs when instance's connection failure is first detected, so the application can inform users of upcoming delays. If failover is successful, the second call occurs when the connection is re-established and usable.

At this time, the client may wish to replay ALTER SESSION statements and inform users that failover has occurred. If failover is unsuccessful, then the callback can be called to inform the application that failover will not occur. If this happens, you can specify that the failover should be re-attempted. Additionally, the callback will be called for each user handle when it attempts to use the connection after failover.

See Also: The *Oracle Call Interface Programmer's Guide.*

Transparent Application Failover Restrictions

When a connection is lost, you will see the following effects:

- All PL/SQL package states on the server are lost at failover.
- ALTER SESSION statements are lost.
- If failover occurs when a transaction is in process, then each subsequent call causes an error message until the user issues an OCITransRollback call. Then an OCI success message is issued. Be sure to check this informational message to see if you must perform any additional operations.
- Continuing work on failed over cursors may cause an error message.
- If the first command after failover is not a SQL SELECT or OCIStmtFetch statement, an error message results.
- Failover only takes effect if the application is programmed using OCI Release 8.0 or greater.
- At failover time, any queries in progress are reissued and processed again from the beginning. This may result in the next query taking a long time if the original query took a long time.

Part V

Parallel Execution

Part Five discusses optimizing parallel execution. The chapters in Part Five are:

26

Tuning Parallel Execution

Parallel execution dramatically reduces response time for data-intensive operations on large databases typically associated with Decision Support Systems (DSS). You can also implement parallel execution on certain types of OLTP (Online Transaction Processing) and hybrid systems.

This chapter explains how to implement parallel execution and tune your system to optimize parallel execution performance.

See Also: *Oracle8i Concepts*, for basic parallel execution concepts and your platform-specific Oracle documentation for more information about tuning parallel execution.

Note: Parallel execution is only available with the Oracle8*i* Enterprise Edition.

This chapter discusses parallel execution in 4 phases. In Phase One, you determine whether to use automated or manual tuning. For many applications, automated tuning provides acceptable performance by automatically setting default values for parameters based on your system configuration.

Phase Two describes how to take advantage of parallelism and partitioning. In Phase Two, you determine the best type parallelism to use based on your needs. This phase also discusses how to take the best advantage of Oracle’s partitioning features.

Phase Three describes how to create, populate, and refresh your database. Phase Four explains how to monitor and fine-tune parallel execution for optimal performance.

The phases and their contents are:

Introduction to Parallel Execution Tuning

Parallel execution is useful for many types of operations accessing significant amounts of data. Parallel execution improves processing for:

- Large table scans and joins
- Creation of large indexes
- Partitioned index scans
- Bulk inserts, updates, and deletes
- Aggregations and copying

You can also use parallel execution to access object types within an Oracle database. For example, you can use parallel execution to access LOBs (Large Binary Objects).

Parallel execution benefits systems if they have *all* of the following characteristics:

- Symmetric Multi-processors (SMP), clusters, or massively parallel systems
- Sufficient I/O bandwidth
- Under utilized or intermittently used CPUs (for example, systems where CPU usage is typically less than 30%)
- Sufficient memory to support additional memory-intensive processes such as sorts, hashing, and I/O buffers

If your system lacks any of these characteristics, parallel execution *may not* significantly improve performance. In fact, parallel execution can reduce system performance on over-utilized systems or systems with small I/O bandwidth.

Note: The term "parallel execution server" designates server processes, or "threads" on NT systems, that perform parallel operations. This is not the same as the Oracle Parallel Server option that refers to multiple Oracle instances accessing the same database.

See Also: For more information about the Oracle Parallel Server, please refer to *Oracle8i Parallel Server Concepts and Administration*.

When to Implement Parallel Execution

Parallel execution provides the greatest performance improvements in Decision Support Systems (DSS). However, Online Transaction Processing (OLTP) systems also benefit from parallel execution, but usually only during batch processing.

During the day, most OLTP systems should probably not use parallel execution. During off-hours, however, parallel execution can effectively process high-volume batch operations. For example, a bank might use parallelized batch programs to perform millions of updates to apply interest to accounts.

The more common example of using parallel execution is for DSS. Complex queries, such as those involving joins of several tables or searches for very large tables, are often best executed in parallel. It is for this reason that the remainder of this chapter discusses parallel execution with an emphasis on DSS environments.

Phase One - Initializing and Tuning Parameters for Parallel Execution

You can initialize and automatically tune parallel execution by setting the parameter PARALLEL_AUTOMATIC_TUNING to TRUE. Once enabled, automated parallel execution controls values for all parameters related to parallel execution. These parameters affect several aspects of server processing, namely, the DOP (degree of parallelism), the adaptive multi-user feature, and memory sizing.

With parallel automatic tuning enabled, Oracle determines parameter settings for each environment based on the number of CPUs on your system and the value set for PARALLEL_THREADS_PER_CPU. The default values Oracle sets for parallel execution processing when PARALLEL_AUTOMATIC_TUNING is TRUE are usually optimal for most environments. In most cases, Oracle's automatically derived settings are at least as effective as manually derived settings.

You can also manually tune parallel execution parameters, however, Oracle recommends using automated parallel execution. Manual tuning of parallel execution is more complex than using automated tuning for two reasons: Manual parallel execution tuning requires more attentive administration than automated tuning, and manual tuning is prone to user load and system resource miscalculations.

Initializing and tuning parallel execution involves the three steps described under the following headings. These steps are:

- Step One: Selecting Automated or Manual Tuning of Parallel Execution
- Step Two: Setting the Degree of Parallelism and Enabling Adaptive Multi-User
- Step Three: Tuning General Parameters

Step Three is a discussion of tuning general parameters. You may find the general parameters information useful if your parallel execution performance requires further tuning after you complete the first two steps.

Several examples describing parallel execution tuning appear at the end of this section. The example scenarios describe configurations that range from completely automated to completely manual systems.

Step One: Selecting Automated or Manual Tuning of Parallel Execution

There are several ways to initialize and tune parallel execution. You can make your environment fully automated for parallel execution, as mentioned, by setting PARALLEL_AUTOMATIC_TUNING to TRUE. You can further customize this type of environment by overriding some of the automatically derived values.

You can also leave PARALLEL_AUTOMATIC_TUNING at its default value of FALSE and manually set the parameters that affect parallel execution. For most OLTP environments and other types of systems that would not benefit from parallel execution, do not enable parallel execution.

Note: Well established, manually tuned systems that achieve desired resource use patterns may not benefit from automated parallel execution.

Automatically Derived Parameter Settings under Fully Automated Parallel Execution

When PARALLEL_AUTOMATIC_TUNING is TRUE, Oracle automatically sets other parameters as shown in Table 26–1. For most systems, you do not need to make further adjustments to have an adequately tuned, fully automated parallel execution environment.

Table 26–1 Parameters Affected by PARALLEL_AUTOMATIC_TUNING

Parameter	Default	Default if PARALLEL_ AUTOMATIC_ TUNING = TRUE	Comments
PARALLEL_ADAPTIVE_ MULTI_USER	FALSE	TRUE	-
PROCESSES	6	The greater of: 1.2 x PARALLEL_ MAX_SERVERS or PARALLEL_MAX_SERVERS + 6 + 5 + (CPUs x 4)	Value is forced up to minimum if PARALLEL_AUTOMATIC_ TUNING is TRUE.
SESSIONS	(PROCESSES x 1.1) + 5	(PROCESSES x 1.1) + 5	Automatic parallel tuning indirectly affects SESSIONS. If you do not set SESSIONS, Oracle sets it based on the value for PROCESSES.

Table 26–1 Parameters Affected by PARALLEL_AUTOMATIC_TUNING

Parameter	Default	Default if PARALLEL_ AUTOMATIC_ TUNING = TRUE	Comments
PARALLEL_MAX_ SERVERS	0	CPU x 10	Use this limit to maximize the number of processes that parallel execution uses. The value for this parameter is port-specific so processing may vary from system to system.
LARGE_POOL_SIZE	None	PARALLEL_ EXECUTION_ POOL+ MTS heap requirements + Backup buffer requests + 600KB	Oracle does not allocate parallel execution buffers from the SHARED_POOL.
PARALLEL_EXECUTION_ MESSAGE_SIZE	2KB (port specific)	4KB (port specific)	Default increased since Oracle allocates memory from the LARGE_POOL.

As mentioned, you can manually adjust the parameters shown in Table 26–1, even if you set PARALLEL_AUTOMATIC_TUNING to TRUE. You might need to do this if you have a highly customized environment or if your system does not perform optimally using the completely automated settings.

Because parallel execution improves performance for a wide range of system types, you might want to use the examples at the end of this section as starting points. After observing your system's performance using these initial settings, you can further customize your system for parallel execution.

Step Two: Setting the Degree of Parallelism and Enabling Adaptive Multi-User

In this step, establish your system's degree of parallelism (DOP) and consider whether to enable adaptive multi-user.

Degree of Parallelism and Adaptive Multi-User and How They Interact

DOP specifies the number of available processes, or threads, used in parallel operations. Each parallel thread may use one or two query processes depending on the query's complexity.

The adaptive multi-user feature adjusts DOP based on user load. For example, you may have a table with a DOP of 5. This DOP may be acceptable with 10 users. But if 10 more users enter the system and you enable the PARALLEL_ADAPTIVE_MULTI_USER feature, Oracle reduces the DOP to spread resources more evenly according to the perceived system load.

Note: Once Oracle determines the DOP for a query, the DOP does not change for the duration of the query.

It is best to use the parallel adaptive multi-user feature when users process simultaneous parallel execution operations. If you enable PARALLEL_AUTOMATIC_TUNING, Oracle automatically sets PARALLEL_ADAPTIVE_MULTI_USER to TRUE.

Note: Disable adaptive multi-user for single-user, batch processing systems or if your system already provides optimal performance.

How the Adaptive Multi-User Algorithm Works

The adaptive multi-user algorithm has several inputs. The algorithm first considers the number of allocated threads as calculated by the database resource manager. The algorithm then considers the default settings for parallelism as set in INIT.ORA, as well as parallelism options used in CREATE TABLE and ALTER TABLE commands and SQL hints.

When a system is overloaded and the input DOP is larger than the default DOP, the algorithm uses the default degree as input. The system then calculates a reduction factor that it applies to the input DOP.

Enabling Parallelism for Tables and Queries

The DOP of tables involved in parallel operations affect the DOP for operations on those tables. Therefore, after setting parallel tuning-related parameters, enable parallel execution for each table you want parallelized using the PARALLEL option of the CREATE TABLE or ALTER TABLE commands. You can also use the PARALLEL hint with SQL statements to enable parallelism for that operation only.

When you parallelize tables, you can also specify the DOP or allow Oracle to set it automatically based on the value of PARALLEL_THREADS_PER_CPU.

Controlling Performance with PARALLEL_THREADS_PER_CPU

The initialization parameter PARALLEL_THREADS_PER_CPU affects algorithms controlling both the DOP and the adaptive multi-user feature. Oracle multiplies the value of PARALLEL_THREADS_PER_CPU by the number of CPUs per instance to derive the number of threads to use in parallel operations.

The adaptive multi-user feature also uses the default DOP to compute the target number of query server processes that should exist in a system. When a system is running more processes than the target number, the adaptive algorithm reduces the DOP of new queries as required. Therefore, you can also use PARALLEL_THREADS_PER_CPU to control the adaptive algorithm.

The default for PARALLEL_THREADS_PER_CPU is appropriate for most systems. However, if your I/O subsystem cannot keep pace with the processors, you may need to increase the value for PARALLEL_THREADS_PER_CPU. In this case, you need more processes to achieve better system scalability. If too many processes are running, reduce the number.

The default for PARALLEL_THREADS_PER_CPU on most platforms is 2. However, the default for machines with relatively slow I/O subsystems can be as high as 8.

Step Three: Tuning General Parameters

This section discusses the following types of parameters:

- Parameters Establishing Resource Limits for Parallel Operations
- Parameters Affecting Resource Consumption
- Parameters Related to I/O

Parameters Establishing Resource Limits for Parallel Operations

The parameters that establish resource limits are:

- PARALLEL_MAX_SERVERS
- PARALLEL_MIN_SERVERS
- LARGE_POOL_SIZE/SHARED_POOL_SIZE
- SHARED_POOL_SIZE
- PARALLEL_MIN_PERCENT
- PARALLEL_SERVER_INSTANCES

PARALLEL_MAX_SERVERS

The recommended value is 2 x *DOP* x *number_of_concurrent_users*.

The PARALLEL_MAX_SEVERS parameter sets a resource limit on the maximum number of processes available for parallel execution. If you set PARALLEL_AUTOMATIC_TUNING to FALSE, you need to manually specify a value for PARALLEL_MAX_SERVERS.

Most parallel operations need at most twice the number of query server processes as the maximum DOP attributed to any table in the operation.

If PARALLEL_AUTOMATIC_TUNING is FALSE, the default value for PARALLEL_MAX_SERVERS is 5. This is sufficient for some minimal operations, but not enough for executing parallel execution. If you manually set the parameter PARALLEL_MAX_SERVERS, set it to 10 times the number of CPUs. This is a reasonable starting value.

To support concurrent users, add more query server processes. You probably want to limit the number of CPU-bound processes to be a small multiple of the number of CPUs: perhaps 4 to 16 times the number of CPUs. This would limit the number of concurrent parallel execution statements to be in the range of 2 to 8.

If a database's users initiate too many concurrent operations, Oracle may not have enough query server processes. In this case, Oracle executes the operations sequentially or displays an error if PARALLEL_MIN_PERCENT is set to another value other than the default value of 0 (zero).

When Users Have Too Many Processes When concurrent users have too many query server processes, memory contention (paging), I/O contention, or excessive context switching can occur. This contention can reduce system throughput to a level lower than if parallel execution were not used. Increase the PARALLEL_MAX_SERVERS value only if your system has sufficient memory and I/O bandwidth for the resulting load. Limiting the total number of query server processes may restrict the number of concurrent users that can execute parallel operations, but system throughput tends to remain stable.

Increasing the Number of Concurrent Users

To increase the number of concurrent users, you could restrict the number of concurrent sessions that resource consumer groups can have. For example:

- You can enable PARALLEL_ADAPTIVE_MULTI_USER
- You can set a large limit for users running batch jobs
- You can set a medium limit for users performing analyses
- You can prohibit a particular class of user from using parallelism

> **See Also:** For more information about resource consumer groups, refer to discussions on the "Database Resource Manager" in the *Oracle8i Administrator's Guide* and *Oracle8i Concepts*.

Limiting the Number of Resources for a User

You can limit the amount of parallelism available to a given user by establishing resource consumer group for the user. Do this to limit the number of sessions, concurrent logons, and the number of parallel processes that any one or group of users can have.

Each query server process working on a parallel execution statement is logged on with a session ID; each process counts against the user's limit of concurrent sessions. For example, to limit a user to 10 processes, set the user's limit to 11. One process is for the parallel coordinator and there remain 10 parallel processes that consist of two sets of query server servers. The user's maximum DOP would thus be 5.

See Also: "Formula for Memory, Users, and Parallel Execution Server Processes" on page 27-2 for more information on balancing concurrent users, DOP, and resources consumed. Also refer to the *Oracle8i Administrator's Guide* for more information about managing resources with user profiles and *Oracle8i Parallel Server Concepts and Administration* for more information on querying GV$ views.

PARALLEL_MIN_SERVERS

The recommended value is 0 (zero).

The system parameter PARALLEL_MIN_SERVERS allows you to specify the number of processes to be started and reserved for parallel operations at startup in a single instance. The syntax is:

```
PARALLEL_MIN_SERVERS=n
```

Where *n* is the number of processes you want to start and reserve for parallel operations.

Setting PARALLEL_MIN_SERVERS balances the startup cost against memory usage. Processes started using PARALLEL_MIN_SERVERS do not exit until the database is shutdown. This way, when a query is issued the processes are likely to be available. It is desirable, however, to recycle query server processes periodically since the memory these processes use can become fragmented and cause the high water mark to slowly increase. When you do not set PARALLEL_MIN_SERVERS, processes exit after they are idle for 5 minutes.

LARGE_POOL_SIZE/SHARED_POOL_SIZE

The following discussion of how to tune the large pool is also true for tuning the shared pool, except as noted under the heading "SHARED_POOL_SIZE" on page 26-17. You must also increase the value for this memory setting by the amount you determine

There is no recommended value for LARGE_POOL_SIZE. Instead, Oracle recommends leaving this parameter unset and having Oracle to set it for you by setting the PARALLEL_AUTOMATIC_TUNING parameter to TRUE. The exception to this is when the system-assigned value is inadequate for your processing requirements.

Note: When PARALLEL_AUTOMATIC_TUNING is set to TRUE, Oracle allocates parallel execution buffers from the large pool. When this parameter is FALSE, Oracle allocates parallel execution buffers from the shared pool.

Oracle automatically computes LARGE_POOL_SIZE if PARALLEL_AUTOMATIC_TUNING is TRUE. To manually set a value for LARGE_POOL_SIZE, query the V$SGASTAT view and increase or decrease the value for LARGE_POOL_SIZE depending on your needs.

For example, if Oracle displays the following error on startup:

```
ORA-27102: out of memory
SVR4 Error: 12: Not enough space
```

Consider reducing the value for LARGE_POOL_SIZE low enough so your database starts. If after lowering the value of LARGE_POOL_SIZE you see the error:

```
ORA-04031: unable to allocate 16084 bytes of shared memory ("large
pool","unknown object","large pool hea","PX msg pool")
```

Execute the following query to determine why Oracle could not allocate the 16,084 bytes:

```
SELECT NAME, SUM(BYTES) FROM V$SGASTAT WHERE POOL='LARGE POOL' GROUP BY
ROLLUP (NAME) ;
```

Oracle should respond with output similar to:

```
NAME                       SUM(BYTES)
-------------------------- ----------
PX msg pool                   1474572
free memory                    562132
                              2036704
3 rows selected.
```

To resolve this, increase the value for LARGE_POOL_SIZE. This example shows the LARGE_POOL_SIZE to be about 2MB. Depending on the amount of memory available, you could increase the value of LARGE_POOL_SIZE to 4MB and attempt to start your database. If Oracle continues to display an ORA-4031 message, gradually increase the value for LARGE_POOL_SIZE until startup is successful.

Computing Additional Memory Requirements for Message Buffers

After you determine the initial setting for the large or shared pool, you must calculate additional memory requirements for message buffers and determine how much additional space you need for cursors.

Adding Memory for Message Buffers You must increase the value for the LARGE_- or the SHARED_POOL_SIZE parameters to accommodate message buffers. The message buffers allow query server processes to communicate with each other. If you enable automatic parallel tuning, Oracle allocates space for the message buffer from the large pool. Otherwise, Oracle allocates space from the shared pool.

Oracle uses a fixed number of buffers per virtual connection between producer and consumer query servers. Connections increase as the square of the DOP increases. For this reason, the maximum amount of memory used by parallel execution is bound by the highest DOP allowed on your system. You can control this value using either the PARALLEL_MAX_SERVERS parameter or by using policies and profiles.

> **See Also:** *Oracle8i Concepts*, for information on how Oracle makes connections between servers.

Calculate how much additional memory you need for message buffers according the following 5 steps. These 5 steps are nearly the same steps Oracle performs when you set the PARALLEL_AUTOMATIC_TUNING parameter to TRUE. If you enable automatic tuning and check the computed value, you will get the same result.

1. Determine the Maximum DOP possible on your system. When determining this value, consider how you parallelize your batch jobs: you use more memory for a single job using a large DOP than you use for multiple jobs with smaller DOPs. Thus, to ensure you have enough memory for message buffers, calculate an "upper bound" DOP. This DOP should also take multiple instances into account. In other words, to use a degree of 4 in 2 instances, the number you calculate should be 8, not 4. A conservative way to compute the maximum value is to take the value of PARALLEL_MAX_SERVERS multiplied by the number of instances and divide by 4. This number is the DOP in the formula appearing after step 5.
2. Determine the number of instances participating in the SQL statements. For most installations, this number will be 1. This value is INSTANCES in the formula.
3. Estimate the maximum number of concurrent queries executing at this DOP. A number of 1 is a reasonable value if either

PARALLEL_ADAPTIVE_MULTI_USER is set to TRUE or if you have set DOP to be a value which is either greater than or equal to the value for PARALLEL_MAX_SERVERS divided by 4. This is because your DOP is then bound by the number of servers. This number is USERS in the formula below.

4. Calculate the maximum number of query server process groups per query. Normally, Oracle uses only one group of query server processes per query. Sometimes with subqueries, however, Oracle uses one group of query server processes for each subquery. A conservative starting value for this number is 2. This number is GROUPS in the formula appearing after step 5.
5. Determine the parallel message size using the value for the parameter PARALLEL_MESSAGE_SIZE. This is usually 2KB or 4KB. Use the SQLPlus SHOW PARAMETERS command to see the current value for PARALLEL_MESSAGE_SIZE.

Memory Formula for SMP Systems Most SMP systems use the following formula:

$$\textit{Memory in bytes} = (3 \times SETS \times USERS \times SIZE \times CONNECTIONS)$$

Where CONNECTIONS = (DOP^2 + 2 x DOP).

Memory Formula for MPP Systems If you are using OPS and the value for INSTANCES is greater than 1, use the following formula. This formula calculates the number of buffers needed for local virtual connections as well as for remote physical connections. You can use the value of REMOTE as the number of remote connections among nodes to help tune your operating system. The formula is:

$$\textit{Memory in bytes} = (GROUPS \times USERS \times SIZE) \times ((LOCAL \times 3) + (REMOTE \times 2))$$

Where:

- CONNECTIONS = (DOP^2 + 2 x DOP)
- LOCAL = CONNECTIONS/INSTANCES
- REMOTE = CONNECTIONS - LOCAL

Each instance uses the memory computed by the formula.

Add this amount to your original setting for the large or shared pool. However, before setting a value for either of these memory structures, you must also consider additional memory for cursors as explained under the following heading.

Calculating Additional Memory for Cursors Parallel execution plans consume more space in the SQL area than serial execution plans. You should regularly monitor shared pool resource use to ensure both structures have enough memory to accommodate your system's processing requirements.

> **See Also:** For more information about execution plans, please refer to Chapter 13, "Using EXPLAIN PLAN". For more information about how query servers communicate, please refer to *Oracle8i Concepts*.

> **Note:** If you used parallel execution in previous releases and now intend to manually tune it, reduce the amount of memory allocated for LARGE_POOL_SIZE to account for the decreased demand on this pool.

Adjusting Memory After Processing Begins

The formulae in this section are just starting points. Whether you are using automated or manual tuning, you should monitor usage on an on-going basis to make sure the size of memory is not too large or too small. To do this, tune the large and shared pools pool after examining the size of structures in the large pool using a query syntax of:

```
   SELECT POOL, NAME, SUM(BYTES) FROM V$SGASTAT WHERE POOL LIKE '%pool%'
   GROUP BY ROLLUP (POOL, NAME);
```

Oracle responds with output similar to:

```
POOL        NAME                       SUM(BYTES)
----------- -------------------------- ----------
large pool  PX msg pool                  38092812
large pool  free memory                    299988
large pool                               38392800
shared pool Checkpoint queue                38496
shared pool KGFF heap                        1964
shared pool KGK heap                         4372
shared pool KQLS heap                     1134432
shared pool LRMPD SGA Table                 23856
shared pool PLS non-lib hp                2096
shared pool PX subheap                  186828
shared pool SYSTEM PARAMETERS            55756
shared pool State objects              3907808
shared pool character set memory         30260
shared pool db_block_buffers            200000
shared pool db_block_hash_buckets        33132
shared pool db_files                    122984
shared pool db_handles                   52416
shared pool dictionary cache            198216
shared pool dlm shared memory          5387924
shared pool enqueue_resources            29016
shared pool event statistics per sess   264768
shared pool fixed allocation callback     1376
shared pool free memory               26329104
shared pool gc_*                         64000
shared pool latch nowait fails or sle    34944
shared pool library cache              2176808
shared pool log_buffer                   24576
shared pool log_checkpoint_timeout       24700
shared pool long op statistics array     30240
shared pool message pool freequeue      116232
shared pool miscellaneous               267624
shared pool processes                    76896
shared pool session param values         41424
shared pool sessions                    170016
shared pool sql area                   9549116
shared pool table columns               148104
shared pool trace_buffers_per_process  1476320
shared pool transactions                 18480
shared pool trigger inform               24684
shared pool                           52248968
                                      90641768
41 rows selected.
```

Evaluate the memory used as shown in your output and alter the setting for LARGE_POOL_SIZE based on your processing needs.

To obtain more memory usage statistics, execute the query:

```
SELECT * FROM V$PX_PROCESS_SYSSTAT WHERE STATISTIC LIKE 'Buffers%';
```

Oracle responds with output similar to:

```
STATISTIC                      VALUE
------------------------------ ----------
Buffers Allocated                   23225
Buffers Freed                       23225
Buffers Current                         0
Buffers HWM                          3620
4 Rows selected.
```

The amount of memory used appears in the statistics "Buffers Current" and "Buffers HWM". Calculate a value for the bytes by multiplying the number of buffers by the value for PARALLEL_EXECUTION_MESSAGE_SIZE. Compare the high water mark to the parallel execution message pool size to determine if you allocated too much memory. For example, in the first output, the value for large pool as shown in ’px msg pool’ is 38092812, or 38MB. The ’Buffers HWM’ from the second output is 3,620, which when multiplied by a parallel execution message size of 4,096 is 14,827,520, or approximately 15MB. In this case, the high water mark has reached approximately 40% of its capacity.

SHARED_POOL_SIZE

As mentioned earlier, if PARALLEL_AUTOMATIC_TUNING is FALSE, Oracle allocates query server processes from the shared pool. In this case, tune the shared pool as described under the previous heading for large pool except for the following:

- Allow for other clients of the shared pool such as shared cursors and stored procedures
- Larger values improve performance in multi-user systems but smaller values use less memory

You must also take into account that using parallel execution generates more cursors. Look at statistics in the V$SQLAREA view to determine how often Oracle recompiles cursors. If the cursor hit ratio is poor, increase the size of the pool.

Use the following query to determine how much memory each component uses and thus to tune the value for SHARED_POOL_SIZE.

```
SELECT POOL, NAME, SUM(BYTES) FROM V$SGASTAT WHERE POOL LIKE ’%pool%’
GROUP BY ROLLUP (POOL, NAME);
```

Oracle responds with output similar to:

```
POOL        NAME                       SUM(BYTES)
----------- -------------------------- ----------
shared pool Checkpoint queue                38496
shared pool KGFF heap                        1320
shared pool KGK heap                         4372
shared pool KQLS heap                      904528
shared pool LRMPD SGA Table                 23856
shared pool PLS non-lib hp                   2096
shared pool PX msg pool                    373864
shared pool PX subheap                      65188
shared pool SYSTEM PARAMETERS               55828
shared pool State objects                 3877520
shared pool character set memory            30260
shared pool db_block_buffers               200000
shared pool db_block_hash_buckets           33132
shared pool db_files                       122984
shared pool db_handles                      36400
shared pool dictionary cache               181792
shared pool dlm shared memory             5387924
shared pool enqueue_resources               25560
shared pool event statistics per sess      189120
shared pool fixed allocation callback        1376
shared pool free memory                  36255072
shared pool gc_*                            64000
shared pool latch nowait fails or sle       34944
shared pool library cache                  559676
shared pool log_buffer                      24576
shared pool log_checkpoint_timeout          24700
shared pool long op statistics array        21600
shared pool message pool freequeue         116232
shared pool miscellaneous                  230016
shared pool network connections             17280
shared pool processes                       53736
shared pool session param values            58684
shared pool sessions                       121440
shared pool sql area                      1232748
shared pool table columns                  148104
shared pool trace_buffers_per_process     1025232
shared pool transactions                    18480
shared pool trigger inform                  16456
shared pool                              51578592
                                         51578592
40 rows selected.
```

You can then monitor the number of buffers used by parallel execution in the same way as explained previously, and compare the 'shared pool PX msg pool' to the current high water mark reported in output from the view V$PX_PROCESS_SYSSTAT.

PARALLEL_MIN_PERCENT

The recommended value for this parameter is 0 (zero).

This parameter allows users to wait for an acceptable DOP depending on the application in use. Setting this parameter to values other than 0 (zero) causes Oracle to return an error when the required minimum DOP cannot be satisfied by the system at a given time.

For example, if you set PARALLEL_MIN_PERCENT to 50, which translates to "50%", and the DOP is reduced because of the adaptive algorithm or because of a resource limitation, then oracle returns ORA-12827. For example:

```
SELECT /*+ PARALLEL(e, 4, 1) */ d.deptno, SUM(SAL)
FROM emp e, dept d WHERE e.deptno = d.deptno
GROUP BY d.deptno ORDER BY d.deptno;
```

Oracle responds with this message:

```
ORA-12827: INSUFFICIENT PARALLEL QUERY SLAVES AVAILABLE
```

PARALLEL_SERVER_INSTANCES

The recommended value is to set this parameter equal to the number of instances in your parallel server environment.

The PARALLEL_SERVER_INSTANCES parameter specifies the number of instances configured in a parallel server environment. Oracle uses the value of this parameter to compute values for LARGE_POOL_SIZE when PARALLEL_AUTOMATIC_TUNING is set to TRUE.

Parameters Affecting Resource Consumption

The parameters that affect resource consumption are:

- HASH_AREA_SIZE
- SORT_AREA_SIZE
- PARALLEL_EXECUTION_MESSAGE_SIZE
- OPTIMIZER_PERCENT_PARALLEL

- PARALLEL_BROADCAST_ENABLE

The first group of parameters discussed in this section affects memory and resource consumption for all parallel operations, and in particular for parallel execution. A second subset of parameters discussed in this section explains parameters affecting parallel DML and DDL. Chapter 27, "Understanding Parallel Execution Performance Issues" describes in detail how these parameters interrelate.

To control resource consumption, configure memory at two levels:

- At the Oracle level, so the system uses an appropriate amount of memory from the operating system.
- At the operating system level for consistency. On some platforms you may need to set operating system parameters that control the total amount of virtual memory available, summed across all processes.

The SGA is typically part of real physical memory. The SGA is static and of fixed size; if you want to change its size, shut down the database, make the change, and restart the database. Oracle allocates the large and shared pools out of the SGA.

A large percentage of the memory used in data warehousing operations is more dynamic. This memory comes from process memory and both the size of process memory and the number of processes can vary greatly. This memory is controlled by the HASH_AREA_SIZE and SORT_AREA_SIZE parameters. Together these parameters control the amount of virtual memory used by Oracle.

Process memory, in turn, comes from virtual memory. Total virtual memory should be somewhat larger than available real memory, which is the physical memory minus the size of the SGA. Virtual memory generally should not exceed twice the size of the physical memory minus the SGA size. If you set virtual memory to a value several times greater than real memory, the paging rate may increase when the machine is overloaded.

As a general rule for memory sizing, each process requires adequate address space for hash joins. A dominant factor in high volume data warehousing operations is the relationship between memory, the number of processes, and the number of hash join operations. Hash joins and large sorts are memory-intensive operations, so you may want to configure fewer processes, each with a greater limit on the amount of memory it can use. Sort performance, however, degrades with increased memory use.

HASH_AREA_SIZE

Set HASH_AREA_SIZE using one of two approaches. The first approach examines how much memory is available after configuring the SGA and calculating the amount of memory processes the system uses during normal loads.

The total amount of memory that Oracle processes are allowed to use should be divided by the number of processes during the normal load. These processes include parallel execution servers. This number determines the total amount of working memory per process. This amount then needs to be shared among different operations in a given query. For example, setting HASH_AREA_SIZE or SORT_AREA_SIZE to half or one third of this number is reasonable.

Set these parameters to the highest number that does not cause swapping. After setting these parameters as described, you should watch for swapping and free memory. If there is swapping, decrease the values for these parameters. If a significant amount of free memory remains, you may increase the values for these parameters.

The second approach to setting HASH_AREA_SIZE requires a thorough understanding of the types of hash joins you execute and an understanding of the amount of data you will be querying against. If the queries and query plans you execute are well understood, this approach is reasonable.

HASH_AREA_SIZE should be approximately half of the square root of *S*, where *S* is the size in megabytes of the smaller of the inputs to the join operation. In any case, the value for HASH_AREA_SIZE should not be less than 1MB.

This relationship can be expressed as follows:

$$HASH\_AREA\_SIZE \;>=\; \frac{\sqrt{S}}{2}$$

For example, if *S* equals 16MB, a minimum appropriate value for HASH_AREA_SIZE might be 2MB summed over all parallel processes. Thus if you have 2 parallel processes, a minimum value for HASH_AREA_SIZE might be 1MB. A smaller hash area is not advisable.

For a large data warehouse, HASH_AREA_SIZE may range from 8MB to 32MB or more. This parameter provides for adequate memory for hash joins. Each process performing a parallel hash join uses an amount of memory equal to HASH_AREA_SIZE.

Hash join performance is more sensitive to HASH_AREA_SIZE than sort performance is to SORT_AREA_SIZE. As with SORT_AREA_SIZE, too large a hash area may cause the system to run out of memory.

The hash area does not cache blocks in the buffer cache; even low values of HASH_AREA_SIZE will not cause this to occur. Too small a setting, however, could adversely affect performance.

HASH_AREA_SIZE is relevant to parallel execution operations and to the query portion of DML or DDL statements.

SORT_AREA_SIZE

The recommended values for this parameter fall in the range from 256KB to 4MB.

This parameter specifies the amount of memory to allocate per query server process for sort operations. If you have a lot of system memory, you can benefit from setting SORT_AREA_SIZE to a large value. This can dramatically increase the performance of sort operations because the entire process is more likely to be performed in memory. However, if memory is a concern for your system, you may want to limit the amount of memory allocated for sort and hash operations.

If the sort area is too small, an excessive amount of I/O is required to merge a large number of sort runs. If the sort area size is smaller than the amount of data to sort, then the sort will move to disk, creating sort runs. These must then be merged again using the sort area. If the sort area size is very small, there will be many runs to merge and multiple passes may be necessary. The amount of I/O increases as SORT_AREA_SIZE decreases.

If the sort area is too large, the operating system paging rate will be excessive. The cumulative sort area adds up quickly because each query server process can allocate this amount of memory for each sort. For such situations, monitor the operating system paging rate to see if too much memory is being requested.

SORT_AREA_SIZE is relevant to parallel execution operations and to the query portion of DML or DDL statements. All CREATE INDEX statements must do some sorting to generate the index. Commands that require sorting include:

- CREATE INDEX
- Direct-load INSERT (if an index is involved)
- ALTER INDEX ... REBUILD

> **See Also:** "HASH_AREA_SIZE" on page 26-21.

PARALLEL_EXECUTION_MESSAGE_SIZE

The recommended value for PARALLEL_EXECUTION_MESSAGE_SIZE is 4KB. If PARALLEL_AUTOMATIC_TUNING is TRUE, the default is 4KB. If PARALLEL_AUTOMATIC_TUNING is FALSE, the default is slightly greater than 2KB.

The PARALLEL_EXECUTION_MESSAGE_SIZE parameter specifies the upper limit for the size of parallel execution messages. The default value is operating system specific and this value should be adequate for most applications. Larger values for PARALLEL_EXECUTION_MESSAGE_SIZE require larger values for LARGE_POOL_SIZE or SHARED_POOL_SIZE, depending on whether you've enabled parallel automatic tuning.

While you may experience significantly improved response time by increasing the value for PARALLEL_EXECUTION_ MESSAGE_SIZE, memory use also drastically increases. For example, if you double the value for PARALLEL_EXECUTION_ MESSAGE_SIZE, parallel execution requires a message source pool that is twice as large.

Therefore, if you set PARALLEL_AUTOMATIC_TUNING to FALSE, then you must adjust the SHARED_POOL_SIZE to accommodate parallel execution messages. If you have set PARALLEL_AUTOMATIC_TUNING to TRUE, but have set LARGE_POOL_SIZE manually, then you must adjust the LARGE_POOL_SIZE to accommodate parallel execution messages.

OPTIMIZER_PERCENT_PARALLEL

The recommended value is 100 / *number_of_concurrent_users.*

This parameter determines how aggressively the optimizer attempts to parallelize a given execution plan. OPTIMIZER_PERCENT_PARALLEL encourages the optimizer to use plans with low response times because of parallel execution, even if total resource used is not minimized.

The default value of OPTIMIZER_PERCENT_PARALLEL is 0 (zero), which, if possible, parallelizes the plan using the fewest resources. Here, the execution time of the operation may be long because only a small amount of resource is used.

Note: Given an appropriate index, Oracle can quickly select a single record from a table; Oracle does not require parallelism to do this. A full scan to locate the single row can be executed in parallel. Normally, however, each parallel process examines many rows. In this case, the response time of a parallel plan will be longer and total system resource use will be much greater than if it were done by a serial plan using an index. With a parallel plan, the delay is shortened because more resources are used. The parallel plan could use up to *n* times more resources where *n* is equal to the value set for the degree of parallelism. A value between 0 and 100 sets an intermediate trade-off between throughput and response time. Low values favor indexes; high values favor table scans.

A nonzero setting of OPTIMIZER_PERCENT_PARALLEL is overridden if you use a FIRST_ROWS hint or set OPTIMIZER_MODE to FIRST_ROWS.

PARALLEL_BROADCAST_ENABLE

The default value is FALSE.

Set this parameter to TRUE if you are joining a very large join result set with a very small result set (size being measured in bytes, rather than number of rows). In this case, the optimizer has the option of broadcasting the small set's rows to each of the query server processes that are processing the rows of the larger set. The result is enhanced performance.

You cannot dynamically set the parameter PARALLEL_BROADCAST_ENABLE as it only affects only hash joins and merge joins.

Parameters Affecting Resource Consumption for Parallel DML and Parallel DDL

The parameters that affect parallel DML and parallel DDL resource consumption are:

- TRANSACTIONS
- ROLLBACK_SEGMENTS
- FAST_START_PARALLEL_ROLLBACK
- LOG_BUFFER
- DML_LOCKS
- ENQUEUE_RESOURCES

Parallel inserts, updates, and deletes require more resources than serial DML operations require. Likewise, PARALLEL CREATE TABLE ... AS SELECT and PARALLEL CREATE INDEX may require more resources. For this reason you may need to increase the value of several additional initialization parameters. These parameters do *not* affect resources for queries.

> **See Also:** *Oracle8i SQL Reference* for complete information about parameters.

TRANSACTIONS

For parallel DML and DDL, each query server process starts a transaction. The parallel coordinator uses the two-phase commit protocol to commit transactions; therefore the number of transactions being processed increases by the DOP. You may thus need to increase the value of the TRANSACTIONS initialization parameter.

The TRANSACTIONS parameter specifies the maximum number of concurrent transactions. The default assumes no parallelism. For example, if you have a DOP of 20, you will have 20 more new server transactions (or 40, if you have two server sets) and 1 coordinator transaction; thus you should increase TRANSACTIONS by 21 (or 41), if they are running in the same instance. If you do not set this parameter, Oracle sets it to 1.1 x SESSIONS.

ROLLBACK_SEGMENTS

The increased number of transactions for parallel DML and DDL requires more rollback segments. For example, one command with a DOP of 5 uses 5 server transactions distributed among different rollback segments. The rollback segments should belong to tablespaces that have free space. The rollback segments should also be unlimited, or you should specify a high value for the MAXEXTENTS parameter of the STORAGE clause. In this way they can extend and not run out of space.

FAST_START_PARALLEL_ROLLBACK

If a system crashes when there are uncommitted parallel DML or DDL transactions, you can speed up transaction recovery during startup by using the FAST_START_PARALLEL_ROLLBACK parameter.

This parameter controls the DOP used when recovering "dead transactions." Dead transactions are transactions that are active before a system crash. By default, the DOP is chosen to be at most two times the value of the CPU_COUNT parameter.

If the default DOP is insufficient, set the parameter to the HIGH. This gives a maximum DOP to be at most 4 times the value of the CPU_COUNT parameter. This feature is available by default.

LOG_BUFFER

Check the statistic "redo buffer allocation retries" in the V$SYSSTAT view. If this value is high, try to increase the LOG_BUFFER size. A common LOG_BUFFER size for a system generating numerous logs is 3 to 5MB. If the number of retries is still high after increasing LOG_BUFFER size, a problem may exist with the disk on which the log files reside. In that case, tune the I/O subsystem to increase the I/O rates for redo. One way of doing this is to use fine-grained striping across multiple disks. For example, use a stripe size of 16KB. A simpler approach is to isolate redo logs on their own disk.

DML_LOCKS

This parameter specifies the maximum number of DML locks. Its value should equal the total of locks on all tables referenced by all users. A parallel DML operation's lock and enqueue resource requirement is very different from serial DML. Parallel DML holds many more locks, so you should increase the value of the ENQUEUE_RESOURCES and DML_LOCKS parameters by equal amounts.

Table 26–2 shows the types of locks acquired by coordinator and query server processes for different types of parallel DML statements. Using this information, you can determine the value required for these parameters. A query server process can work on one or more partitions or subpartitions, but a partition or subpartition can only be worked on by one server process (this is different from parallel execution).

Table 26–2 Locks Acquired by Parallel DML Statements

Type of statement	Coordinator process acquires:	Each parallel execution server acquires:
Parallel UPDATE or DELETE into partitioned table; WHERE clause pruned to a subset of partitions/subpartitions	1 table lock SX 1 partition lock X per pruned (sub)partition	1 table lock SX 1 partition lock NULL per pruned (sub)partition owned by the query server process 1 partition-wait lock S per pruned (sub)partition owned by the query server process

Table 26–2 Locks Acquired by Parallel DML Statements

Type of statement	Coordinator process acquires:	Each parallel execution server acquires:
Parallel row-migrating UPDATE into partitioned table; WHERE clause pruned to a subset of (sub)partitions	1 table lock SX	1 table lock SX
	1 partition X lock per pruned (sub)partition	1 partition lock NULL per pruned (sub)partition owned by the query server process
		1 partition-wait lock S per pruned partition owned by the query server process
	1 partition lock SX for all other (sub)partitions	1 partition lock SX for all other (sub)partitions
Parallel UPDATE, DELETE, or INSERT into partitioned table	1 table lock SX	1 table lock SX
	Partition locks X for all (sub)partitions	1 partition lock NULL per (sub)partition owned by the query server process
		1 partition-wait lock S per (sub)partition owned by the query server process
Parallel INSERT into nonpartitioned table	1 table lock X	None

Note: Table, partition, and partition-wait DML locks all appear as TM locks in the V$LOCK view.

Consider a table with 600 partitions running with a DOP of 100. Assume all partitions are involved in a parallel UPDATE/DELETE statement with no row-migrations.

The coordinator acquires:	1 table lock SX.
	600 partition locks X.
Total server processes acquire:	100 table locks SX.
	600 partition locks NULL.
	600 partition-wait locks S.

ENQUEUE_RESOURCES

This parameter sets the number of resources that can be locked by the lock manager. Parallel DML operations require many more resources than serial DML. Therefore, increase the value of the ENQUEUE_RESOURCES and DML_LOCKS parameters by equal amounts.

See Also: "DML_LOCKS" on page 26-26.

Parameters Related to I/O

The parameters that affect I/O are:

- DB_BLOCK_BUFFERS
- DB_BLOCK_SIZE
- DB_FILE_MULTIBLOCK_READ_COUNT
- HASH_MULTIBLOCK_IO_COUNT
- SORT_MULTIBLOCK_READ_COUNT
- DISK_ASYNCH_IO and TAPE_ASYNCH_IO

These parameters also affect the optimizer which ensures optimal performance for parallel execution I/O operations.

DB_BLOCK_BUFFERS

When you perform parallel updates and deletes, the buffer cache behavior is very similar to any system running a high volume of updates. For more information, see "Tuning the Buffer Cache" on page 19-25.

DB_BLOCK_SIZE

The recommended value is 8KB or 16KB.

Set the database block size when you create the database. If you are creating a new database, use a large block size.

DB_FILE_MULTIBLOCK_READ_COUNT

The recommended value is 8 for 8KB block size, or 4 for 16KB block size.

This parameter determines how many database blocks are read with a single operating system READ call. The upper limit for this parameter is platform-dependent. If you set DB_FILE_MULTIBLOCK_READ_COUNT to an excessively high value, your operating system will lower the value to the highest allowable level when you start your database. In this case, each platform uses the highest value possible. Maximum values generally range from 64KB to 1MB.

HASH_MULTIBLOCK_IO_COUNT

The recommended value is 4.

This parameter specifies how many blocks a hash join reads and writes at once. Increasing the value of HASH_MULTIBLOCK_IO_COUNT decreases the number of hash buckets. If a system is I/O bound, you can increase the efficiency of I/O by having larger transfers per I/O.

Because memory for I/O buffers comes from the HASH_AREA_SIZE, larger I/O buffers mean fewer hash buckets. There is a trade-off, however. For large tables (hundreds of gigabytes in size) it is better to have more hash buckets and slightly less efficient I/Os. If you find an I/O bound condition on temporary space during hash joins, consider increasing the value of HASH_MULTIBLOCK_IO_COUNT.

SORT_MULTIBLOCK_READ_COUNT

The recommended value is to use the default value.

The SORT_MULTIBLOCK_READ_COUNT parameter specifies the number of database blocks to read each time a sort performs a read from a temporary segment. Temporary segments are used by a sort when the data does not fit in SORT_AREA_SIZE of memory.

If the system is performing too many I/Os per second during sort operations and the CPUs are relatively idle during that time, consider increasing the SORT_MUTLIBLOCK_READ_COUNT parameter to force the sort operations to perform fewer, larger I/Os.

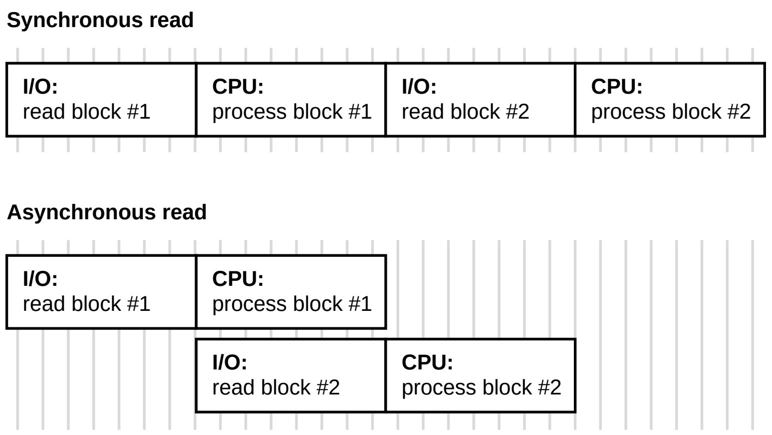
> **See Also:** For more information, please see "Tuning Sorts" on page 20-34.

DISK_ASYNCH_IO and TAPE_ASYNCH_IO

The recommended value is TRUE.

These parameters enable or disable the operating system’s asynchronous I/O facility. They allow query server processes to overlap I/O requests with processing when performing table scans. If the operating system supports asynchronous I/O, leave these parameters at the default value of TRUE.

Figure 26–1 Asynchronous Read



Asynchronous operations are currently supported for parallel table scans, hash joins, sorts, and serial table scans. However, this feature may require operating system specific configuration and may not be supported on all platforms. Check your Oracle operating system specific documentation.

Example Parameter Setting Scenarios for Parallel Execution

The following examples describe a limited variety of parallel execution implementation possibilities. Each example begins by using either automatic or manual parallel execution tuning. Oracle automatically sets other parameters based on each sample system's characteristics and on how parallel execution tuning was initialized. The examples then describe setting the degree of parallelism and the enabling of the adaptive multi-user feature.

The effects that the parameter settings in these examples have on internally-derived settings and overall performance are only approximations. Your system's performance characteristics will vary depending on operating system dependencies and user workloads.

With additional adjustments, you can fine tune these examples to make them more closely resemble your environment. To further analyze the consequences of setting PARALLEL_AUTOMATIC_TUNING to TRUE, refer to Table 26–1 on page 26-5.

In your production environment, after you set the DOP for your tables and enable the adaptive multi-user feature, you may want to analyze system performance as explained in "Phase Four - Monitoring Parallel Execution Performance" on page 26-78. If your system performance remains poor, refer to Phase One's explanation of "Step Three: Tuning General Parameters" on page 26-9.

The following four examples describe different system types in ascending order of size and complexity.

Example One: Small Datamart

In this example, the DBA has limited parallel execution experience and does not have time to closely monitor the system.

The database is mostly a star type schema with some summary tables and a few tables in third normal form. The workload is mostly "ad hoc" in nature. Users expect parallel execution to improve the performance of their high-volume queries.

Other facts about the system are:

- CPUS = 4
- Main Memory = 750MB
- Disk = 40GB
- Users = 16

The DBA makes the following settings:

- PARALLEL_AUTOMATIC_TUNING = TRUE
- SHARED_POOL_SIZE = 12MB
- TRANSACTIONS = Left unset to use system default

Oracle automatically makes the following default settings:

- PARALLEL_MAX_SERVERS = 64
- PARALLEL_ADAPTIVE_MULTI_USER = TRUE
- PARALLEL_THREADS_PER_CPU = 2
- PROCESSES = 76
- SESSIONS = 88
- TRANSACTIONS = 96
- LARGE_POOL_SIZE = 29MB

Parameter Settings for DOP and the Adaptive Multi-User Feature

The DBA parallelizes every table having more than 10,000 rows using a command similar to the following:

```
ALTER TABLE employee PARALLEL;
```

In this example, because PARALLEL_THREADS_PER_CPU is 2 and the number of CPUs is 4, the DOP is 8. Because PARALLEL_ADAPTIVE_MULTI_USER is set to TRUE, Oracle may reduce this DOP in response to the system load that exists at the time of the query's initiation.

Example Two: Medium-sized Data Warehouse

In this example the DBA is experienced but is also busy with other responsibilities. The DBA knows how to organize users into resource consumer groups and uses views and other roles to control access to parallelism. The DBA also has experimented with manually adjusting the settings that automated parallel tuning generates and has chosen to use all of them except for the PARALLEL_ADAPTIVE_MULTI_USER parameter which the DBA sets to FALSE.

The system workload involves some adhoc queries and a high volume of batch operations to convert a central repository into summary tables and star schemas. Most queries on this system are generated by Oracle Express and other tools.

The database has source tables in third normal form and end-user tables in a star schema and summary form only.

Other facts about the system are:

- CPUS = 8
- Main Memory = 2GB
- Disk = 80GB
- Users = 40

The DBA makes the following settings:

- PARALLEL_AUTOMATIC_TUNING = TRUE
- PARALLEL_ADAPTIVE_MULTI_USER = FALSE
- PARALLEL_THREADS_PER_CPU = 4
- SHARED_POOL_SIZE = 20MB

The DBA also sets other parameters unrelated to parallelism. As a result, Oracle responds by automatically adjusting the following parameter settings:

- PROCESSES = 307
- SESSIONS = 342
- TRANSACTIONS = 376
- PARALLEL_MAX_SERVERS = 256
- LARGE_POOL_SIZE = 78MB

Parameter Settings for DOP and the Adaptive Multi-User Feature

The DBA parallelizes some tables in the data warehouse while creating other views for special users:

```
ALTER TABLE sales PARALLEL;
CREATE VIEW invoice_parallel AS SELECT /*+ PARALLEL(P) */ * FROM invoices P;
```

The DBA allows the system to use the PARALLEL_THREADS_PER_CPU setting of 4 with 8 CPUs. The DOP for the tables is 32. This means a simple query uses 32 processes while more complex queries use 64.

Example Three: Large Data Warehouse

In this example, the DBA is experienced and is occupied primarily with managing this system. The DBA has good control over resources and understands how to tune the system. The DBA schedules large queries in batch mode.

The workload includes some adhoc parallel queries. As well, a large number of serial queries are processed against a star schema. There is also some batch processing that generates summary tables and indexes. The database is completely denormalized and the Oracle Parallel Server option is in use.

Other facts about the system are:

- 24 Nodes, 1 CPU per node
- Uses MPP Architecture (Massively Parallel Processing)
- Main Memory = 750MB per node
- Disk = 200GB
- Users = 256

The DBA uses manual parallel tuning by setting the following:

- PARALLEL_AUTOMATIC_TUNING = FALSE
- PARALLEL_THREADS_PER_CPU = 1
- PARALLEL_MAX_SERVERS = 10
- SHARED_POOL_SIZE = 75MB
- PARALLEL_SERVER_INSTANCES = 24
- PARALLEL_SERVER = TRUE
- PROCESSES = 40
- SESSIONS = 50
- TRANSACTIONS = 60

The DBA also sets other parameters unrelated to parallel execution. Because PARALLEL_AUTOMATIC_TUNING is set to FALSE, Oracle allocates parallel execution buffers from the SHARED_POOL.

Parameter Settings for DOP and the Adaptive Multi-User Feature

The DBA parallelizes tables in the data warehouse by explicitly setting the DOP using syntax similar to the following:

```
ALTER TABLE department1 PARALLEL 10;
ALTER TABLE department2 PARALLEL 5;
CREATE VIEW current_sales AS SELECT /*+ PARALLEL(P, 20) */ * FROM sales P;
```

In this example, Oracle does not make calculations for parallel execution because the DBA has manually set all parallel execution parameters.

EXAMPLE Four: Very Large Data Warehouse

In this example, the DBA is very experienced and is dedicated to administering this system. The DBA has good control over the environment, but the variety of users requires the DBA to devote constant attention to the system.

The DBA sets PARALLEL_AUTOMATIC_TUNING to TRUE which makes Oracle allocate parallel execution buffers from the large pool. PARALLEL_ADAPTIVE_MULTI_USER is automatically enabled. After gaining experience with the system, the DBA fine-tunes the system supplied defaults to further improve performance.

The database is a very large data warehouse with data marts residing on the same machine. The data marts are generated and refreshed from data in the warehouse.

The warehouse is mostly normalized while the marts are mostly star schemas and summary tables. The DBA has carefully customized system parameters through experimentation.

Other facts about the system are:

- CPUS = 64
- Main Memory 32GB
- Disk = 3TB
- Users = 1,000

The DBA makes the following settings:

- PARALLEL_AUTOMATIC_TUNING = TRUE
- PARALLEL_MAX_SERVERS = 600
- PARALLEL_MIN_SERVER = 600
- LARGE_POOL_SIZE = 1,300MB
- SHARED_POOL_SIZE = 500MB
- PROCESSES = 800
- SESSIONS = 900
- TRANSACTIONS = 1,024

Parameter Settings for DOP and the Adaptive Multi-User Feature

The DBA has carefully evaluated which users and tables require parallelism and has set the values according to their requirements. The DBA has taken all steps mentioned in the earlier examples, but in addition, the DBA also uses the following command during peak user hours to enable the adaptive DOP algorithms:

```
ALTER SYSTEM SET PARALLEL_ADAPTIVE_MULTI_USER = TRUE;
```

During off hours when batch processing is about to begin, the DBA disables adaptive processing by issuing the command:

```
ALTER SYSTEM SET PARALLEL_ADAPTIVE_MULTI_USER = FALSE;
```

Phase Two - Tuning Physical Database Layouts for Parallel Execution

This section describes how to tune the physical database layout for optimal performance of parallel execution.

- Types of Parallelism
- Striping Data
- Partitioning Data
- Determining the Degree of Parallelism
- Populating the Database Using Parallel Load
- Setting Up Temporary Tablespaces for Parallel Sort and Hash Join
- Creating Indexes in Parallel
- Additional Considerations for Parallel DML Only

Types of Parallelism

Different parallel operations use different types of parallelism. The optimal physical database layout depends on what parallel operations are most prevalent in your application.

The basic unit of parallelism is a called a *granule*. The operation being parallelized (a table scan, table update, or index creation, for example) is divided by Oracle into granules. Query server processes execute the operation one granule at a time. The number of granules and their size affect the DOP (degree of parallelism) you can use. It also affects how well the work is balanced across query server processes.

Block Range Granules

Block range granules are the basic unit of most parallel operations. This is true even on partitioned tables; it is the reason why, on Oracle, the parallel degree is not related to the number of partitions.

Block range granules are ranges of physical blocks from a table. Because they are based on physical data addresses, Oracle can size block range granules to allow better load balancing. Block range granules permit dynamic parallelism that does not depend on static pre-allocation of tables or indexes. On SMP (symmetric multi-processors) systems, granules are located on different devices to drive as many disks as possible. On many MPP (massively parallel processing) systems, block range granules are preferentially assigned to query server processes that have physical proximity to the disks storing the granules. Block range granules are also used with global striping.

When block range granules are used predominantly for parallel access to a table or index, there are administrative considerations such as recovery or using partitions for deleting portions of data that may influence partition layout more than performance considerations. The number of disks that you stripe partitions over should be at least equal to the value of the DOP so that parallel execution operations can take advantage of partition pruning.

> **See Also:** For MPP systems, see your platform-specific documentation.

Partition Granules

When partition granules are used, a query server process works on an entire partition or subpartition of a table or index. Because partition granules are statically determined when a table or index is created, partition granules do not allow as much flexibility in parallelizing an operation. This means that the allowable DOP might be limited, and that load might not be well balanced across query server processes.

Partition granules are the basic unit of parallel index range scans and parallel operations that modify multiple partitions of a partitioned table or index. These operations include parallel update, parallel delete, parallel direct-load insert into partitioned tables, parallel creation of partitioned indexes, and parallel creation of partitioned tables. Operations such as parallel DML and CREATE LOCAL INDEX, do not recognize block range granules.

When partition granules are used for parallel access to a table or index, it is important that there be a relatively large number of partitions (at least three times the DOP), so Oracle can effectively balance work across the query server processes.

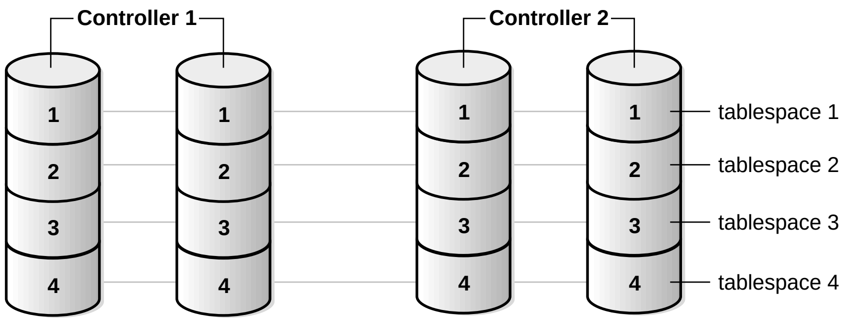
> **See Also:** *Oracle8i Concepts* for information on disk striping and partitioning.

Striping Data

To avoid I/O bottlenecks during parallel processing, tablespaces accessed by parallel operations should be striped. As shown in Figure 26–2, tablespaces should always stripe *over at least as many devices as CPUs;* in this example, there are four CPUs. As was mentioned for partitioned granules, the number of disks over which you stripe these tablespaces should be at least equal to the value set for DOP.

Stripe tablespaces for tables, tablespaces for indexes, rollback segments, and temporary tablespaces. You must also spread the devices over controllers, I/O channels, and/or internal busses.

Figure 26–2 Striping Objects Over at Least as Many Devices as CPUs



To stripe data during loads, use the FILE= clause of parallel loader to load data from multiple load sessions into different files in the tablespace. To make striping effective, ensure that enough controllers and other I/O components are available to support the bandwidth of parallel data movement into and out of the striped tablespaces.

The operating system or volume manager can perform striping (OS striping), or you can perform striping manually for parallel operations.

We recommend using a large stripe size of at least 64KB with OS striping when possible. This approach always performs better than manual striping, especially in multi-user environments.

Operating System Striping Operating system striping is usually flexible and easy to manage. It supports multiple users running sequentially as well as single users running in parallel. Two main advantages make OS striping preferable to manual striping, unless the system is very small or availability is the main concern:

- For parallel scan operations (such as full table scan or fast full scan), operating system striping increases the number of disk seeks. Nevertheless, this is largely compensated by the large I/O size (DB_BLOCK_SIZE * MULTIBLOCK_READ_COUNT) that should enable this operation to reach the maximum I/O throughput for your platform. This maximum is in general limited by the number of controllers or I/O buses of the platform, not by the number of disks (unless you have a very small configuration).

- For index probes (for example, within a nested loop join or parallel index range scan), operating system striping enables you to avoid hot spots: I/O is then more evenly distributed across the disks.

Stripe size must be at least as large as the I/O size. If stripe size is larger than I/O size by a factor of 2 or 4, then certain tradeoffs may arise. The large stripe size can be beneficial because it allows the system to perform more sequential operations on each disk; it decreases the number of seeks on disk. The disadvantage is that it reduces the I/O parallelism so fewer disks are simultaneously active. If you encounter problems, increase the I/O size of scan operations (going, for example, from 64KB to 128KB), instead of changing the stripe size. The maximum I/O size is platform-specific (in a range, for example, of 64KB to 1MB).

With OS striping, from a performance standpoint, the best layout is to stripe data, indexes, and temporary tablespaces across all the disks of your platform. In this way, maximum I/O performance (both in terms of throughput and number of I/Os per second) can be reached when one object is accessed by a parallel operation. If multiple objects are accessed at the same time (as in a multi-user configuration), striping automatically limits the contention. If availability is a major concern, associate this method with hardware redundancy, for example RAID5, which permits both performance and availability.

Manual Striping You can use manual striping on all platforms. To do this, add multiple files to each tablespace, each on a separate disk. If you use manual striping correctly, your system will experience significant performance gains. However, you should be aware of several drawbacks that may adversely affect performance if you do not stripe correctly.

First, when using manual striping, the DOP is more a function of the number of disks than of the number of CPUs. This is because it is necessary to have one server process per datafile to drive all the disks and limit the risk of experiencing I/O bottlenecks. Also, manual striping is very sensitive to datafile size skew which can affect the scalability of parallel scan operations. Second, manual striping requires more planning and set up effort that operating system striping.

> **See Also:** *Oracle8i Concepts* for information on disk striping and partitioning. For MPP systems, see your platform-specific Oracle documentation regarding the advisability of disabling disk affinity when using operating system striping.

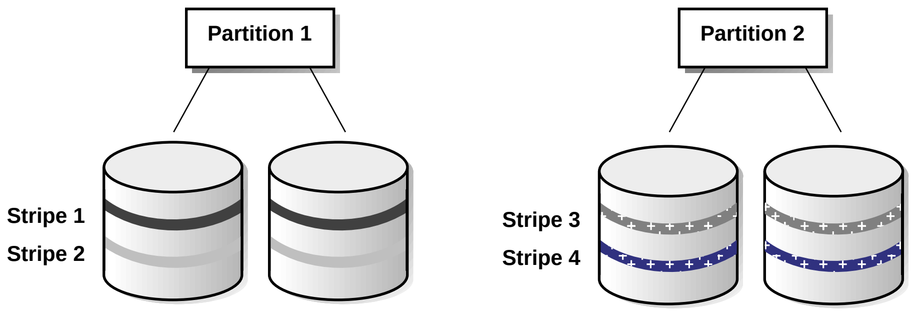
Local and Global Striping

Local striping, which applies only to partitioned tables and indexes, is a form of non-overlapping disk-to-partition striping. Each partition has its own set of disks and files, as illustrated in Table 26–3. There is no overlapping disk access, and no overlapping of files.

An advantage of local striping is that if one disk fails, it does not affect other partitions. Moreover, you still have some striping even if you have data in only one partition.

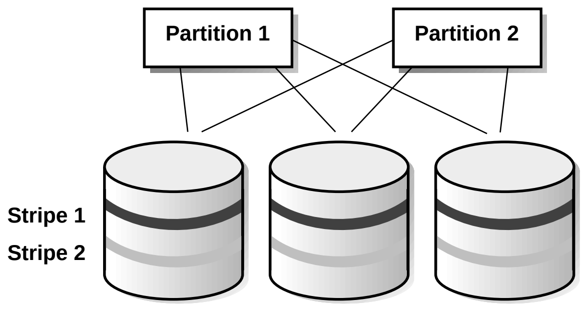
A disadvantage of local striping is that you need many more disks to implement it—each partition requires a few disks of its own. Another major disadvantage is that after partition pruning to only a single or a few partitions, the system will have limited I/O bandwidth. As a result, local striping is not very practical for parallel operations. For this reason, consider local striping only if your main concern is availability, and not parallel execution. A good compromise might be to use global striping associated with RAID5, which permits both performance and availability.

Figure 26–3 Local Striping



Global striping, illustrated in Figure 26–4, entails overlapping disks and partitions.

Figure 26–4 Global Striping



Global striping is advantageous if you have partition pruning and need to access data only in one partition. Spreading the data in that partition across many disks improves performance for parallel execution operations. A disadvantage of global striping is that if one disk fails, all partitions are affected.

> **See Also:** "Striping and Media Recovery" on page 26-44.

Analyzing Striping

There are two considerations when analyzing striping issues for your applications. First, consider the cardinality of the relationships among the objects in a storage system. Second, consider what you can optimize in your striping effort: full table scans, general tablespace availability, partition scans, or some combinations of these goals. These two topics are discussed under the following headings.

Cardinality of Storage Object Relationships To analyze striping, consider the following relationships:

Figure 26–5 Cardinality of Relationships

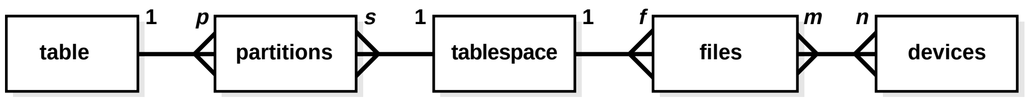


Figure 26–5 shows the cardinality of the relationships among objects in a typical Oracle storage system. For every table there may be:

- *p* partitions, shown in Figure 26–5 as a one-to-many relationship
- *s* partitions for every tablespace, shown in Figure 26–5 as a many-to-one relationship
- *f* files for every tablespace, shown in Figure 26–5 as a one-to-many relationship
- *m* files to *n* devices, shown in Figure 26–5 as a many-to-many relationship

Goals. You may wish to stripe an object across devices to achieve one of three goals:

- Goal 1: To optimize full table scans. This means placing a table on many devices.
- Goal 2: To optimize availability. This means restricting the tablespace to a few devices.
- Goal 3: To optimize partition scans. This means achieving intra-partition parallelism by placing each partition on many devices.

To attain both Goal 1 and Goal 2, having the table reside on many devices, with the highest possible availability, you can maximize the number of partitions *p* and minimize the number of partitions per tablespace *s*.

For highest availability but the least intra-partition parallelism, place each partition in its own tablespace. Do not used striped files, and use one file per tablespace. To minimize Goal 2 and thereby minimize availability, set *f* and *n* equal to 1.

When you minimize availability you maximize intra-partition parallelism. Goal 3 conflicts with Goal 2 because you cannot simultaneously maximize the formula for Goal 3 and minimize the formula for Goal 2. You must compromise to achieve some benefits of both goals.

Goal 1: To optimize full table scans. Having a table on many devices is beneficial because full table scans are scalable.

Calculate the number of partitions multiplied by the number of files in the tablespace multiplied by the number of devices per file. Divide this product by the number of partitions that share the same tablespace, multiplied by the number of files that share the same device. The formula is as follows:

$$\textit{Number of devices per table} = \frac{p \times f \times n}{s \times m}$$

You can do this by having *t* partitions, with every partition in its own tablespace, if every tablespace has one file, and these files are not striped.

t x 1 / p x 1 x 1, up to t devices

If the table is not partitioned, but is in one tablespace in one file, stripe it over *n* devices.

1 x 1 x n devices

Maximum *t* partitions, every partition in its own tablespace, *f* files in each tablespace, each tablespace on a striped device:

t x f x n devices

Goal 2: To optimize availability. Restricting each tablespace to a small number of devices and having as many partitions as possible helps you achieve high availability.

$$\text{Number of devices per tablespace} = \frac{f \times n}{m}$$

Availability is maximized when *f* = *n* = *m* = 1 and *p* is much greater than 1.

Goal 3: To optimize partition scans. Achieving intra-partition parallelism is beneficial because partition scans are scalable. To do this, place each partition on many devices.

$$\text{Number of devices per partition} = \frac{f \times n}{s \times m}$$

Partitions can reside in a tablespace that can have many files. There could be either

- Many files per tablespace or
- Striped file

Striping and Media Recovery

Striping affects media recovery. Loss of a disk usually means loss of access to all objects stored on that disk. If all objects are striped over all disks, then loss of any disk stops the entire database. Furthermore, you may need to restore all database files from backups, even if each file has only a small fraction of its total data stored on the failed disk.

Often, the same OS subsystem that provides striping also provides mirroring. With the declining price of disks, mirroring can provide an effective supplement to backups and log archival--*but not a substitute for them*. Mirroring can help your system recover from device failures more quickly than with a backup, but is not as robust. Mirroring does not protect against software faults and other problems that an independent backup would protect your system against.

You can effectively use mirroring if you are able to reload read-only data from the original source tapes. If you have a disk failure, restoring data from backups could involve lengthy downtime, whereas restoring it from a mirrored disk would enable your system to get back online quickly.

RAID technology is even less expensive than mirroring. RAID avoids full duplication in favor of more expensive write operations. For "read-mostly" applications, this may suffice.

Note: RAID technology is particularly slow on write operations. This slowness may affect your database restore time to a point that RAID performance is unacceptable.

See Also: For a discussion of manually striping tables across datafiles, refer to "Striping Disks" on page 20-22. For a discussion of media recovery issues, see "Backup and Recovery of the Data Warehouse" on page 11-9.

For more information about automatic file striping and tools you can use to determine I/O distribution among your devices, refer to your operating system documentation.

Partitioning Data

This section describes the partitioning features that significantly enhance data access and greatly improve overall applications performance. This is especially true for applications accessing tables and indexes with millions of rows and many gigabytes of data.

Partitioned tables and indexes facilitate administrative operations by allowing these operations to work on subsets of data. For example, you can add a new partition, organize an existing partition, or drop a partition with less than a second of interruption to a read-only application.

Using the partitioning methods described in this section can help you tune SQL statements to avoid unnecessary index and table scans (using partition pruning). You can also improve the performance of massive join operations when large amount of data, for example, several millions rows, are joined together; do this using partition-wise joins. Finally, partitioning data greatly improves manageability of very large databases and dramatically reduces the time required for administrative tasks such as backup and restore.

Types of Partitioning

Oracle offers three partitioning methods:

- Range
- Hash
- Composite

Each partitioning method has a different set of advantages and disadvantages. Thus, each method is appropriate for a particular situation where the others are not.

> **See Also:** *Oracle8i Concepts*, for more information on partitioning.

Range Partitioning Range partitioning maps data to partitions based on boundaries identified by ranges of column values that you establish for each partition. This method is useful primarily for DSS applications that manage historical data.

Hash Partitioning Hash partitioning maps data to partitions based on a hashing algorithm that Oracle applies to a partitioning key identified by the user. The hashing algorithm evenly distributes rows among partitions. Therefore, the resulting set of partitions should be approximately of the same size. This also makes hash partitioning ideal for distributing data evenly across devices. Hash partitioning is also a good and easy-to-use alternative to range partitioning when data is not historical in content.

> **Note:** You cannot create alternate hashing algorithms.

Composite Partitioning Composite partitioning combines the features of range and hash partitioning. With composite partitioning, Oracle first distributes data into partitions according to boundaries established by the partition ranges. Then Oracle further divides the data into subpartitions within each range partition. Oracle uses a hashing algorithm to distribute data into the subpartitions.

> **See Also:** For more information, please refer to, and *Oracle8i Concepts.* and the *Oracle8i Administrator's Guide.*

Index Partitioning

You can create both local and global indexes on a table partitioned by range, hash, or composite. Local indexes inherit the partitioning attributes of their related tables.

For example, if you create a local index on a composite table, Oracle automatically partitions the local index using the composite method.

Oracle only supports range partitioning for global indexes. Therefore, you cannot partition global indexes using the hash or composite partitioning methods.

Performance Issues for Range, Hash and Composite Partitioning

The following section describes performance issues for range, hash, and composite partitioning.

Performance Considerations for Range Partitioning As mentioned, range partitioning is a convenient method for partitioning historical data. The boundaries of range partitions define the ordering of the partitions in the tables or indexes.

The most common use of range partitioning leverages the partitioning of data into time intervals on a column of type “date”. Because of this, SQL statements accessing range partitions tend to focus on timeframes. An example of this is a SQL statement similar to "select data from a particular period in time". In such a scenario, if each partition represents one month’s worth of data, the query "find data of month 98-DEC" needs to access only the December partition of year 98. This reduces the amount of data scanned to a fraction of the total data available. This optimization method is called ’partition pruning’.

Range partitioning is also ideal when you periodically load new data and purge old data. This ’adding’ or ’dropping’ of partitions is a major manageability enhancement.

It is common to keep a ’rolling window’ of data, for example keeping the last 36 months of data online. Range partitioning simplifies this process: to add a new month’s data you load it into a separate table, clean the data, index it, and then add it to the range partitioned table using the EXCHANGE PARTITION command; all while the table remains online. Once you add the new partition, you can drop the ’trailing’ month with the DROP PARTITION command.

In conclusion, consider using Range partitioning when:

- Very large tables are frequently scanned by a range predicate on a column that is a good partitioning column, such as ORDER_DATE or PURCHASE_DATE. Partitioning the table on that column would enable partitioning pruning.
- You want to maintain a ’rolling window’ of data
- You cannot complete administrative operations on large tables, such as backup and restore, in an allotted timeframe

- You need to implement parallel DML (PDML) operations

The following SQL example creates the table “Sales” for a period of two years, 1994 and 1995, and partitions it by range according to the column s_saledate to separate the data into eight quarters, each corresponding to a partition:

```
CREATE TABLE sales
(s_productid NUMBER,
s_saledate DATE,
s_custid NUMBER,
s_totalprice NUMBER)
PARTITION BY RANGE(s_saledate)
(PARTITION sal94q1 VALUES LESS THAN TO_DATE (01-APR-1994, DD-MON-YYYY),
PARTITION sal94q2 VALUES LESS THAN TO_DATE (01-JUL-1994, DD-MON-YYYY),
PARTITION sal94q3 VALUES LESS THAN TO_DATE (01-OCT-1994, DD-MON-YYYY),
PARTITION sal94q4 VALUES LESS THAN TO_DATE (01-JAN-1995, DD-MON-YYYY),
PARTITION sal95q1 VALUES LESS THAN TO_DATE (01-APR-1995, DD-MON-YYYY),
PARTITION sal95q2 VALUES LESS THAN TO_DATE (01-JUL-1995, DD-MON-YYYY),
PARTITION sal95q3 VALUES LESS THAN TO_DATE (01-OCT-1995, DD-MON-YYYY),
PARTITION sal95q4 VALUES LESS THAN TO_DATE (01-JAN-1996, DD-MON-YYYY));
```

Performance Considerations for Hash Partitioning The way Oracle distributes data in hashed partitions has no logical meaning. Therefore, hash partitioning is not an effective way to manage historical data. However, hashed partitions share all other performance characteristics of range partitions. This means using partition pruning is limited to equality predicates. You can also use partition-wise joins, parallel index access and PDML.

> **See Also:** Partition-wise joins are described later in this chapter under the heading "Partition-wise Joins" on page 26-52.

As a general rule, use hash partitioning:

- To partition data, for example, to improve the availability and manageability of large tables or to enable PDML, but your table does not store historical data so range partitioning is not appropriate
- To avoid data skew among partitions. Hash partitioning is an effective means of distributing data because Oracle hashes the data into a number of partitions, each of which can reside on a separate device. Thus, data is evenly spread over as many devices as required to maximize I/O throughput. Similarly, you can use hash partitioning to evenly distribute data among the nodes of an MPP platform that uses the Oracle Parallel Server.

- If it is important to use partition pruning and partition-wise joins according to a partitioning key.

> **Note:** In hash partitioning, partition pruning is limited to using equality or inlist predicates.

If you add or coalesce a hashed partition, Oracle automatically re-arranges the rows to reflect the change in the number of partitions and subpartitions. The hash function that Oracles uses is especially designed to limit the cost of this reorganization. Instead of reshuffling all the rows in the table, Oracles uses an 'add partition' logic that splits one and only one of the existing hashed partitions. Conversely, Oracle coalesces a partition by merging two existing hashed partitions.

Although this dramatically improves the manageability of hash partitioned tables, it means that the hash function can cause a skew if the number of partitions of a hash partitioned table, or the number of subpartitions in each partition of a composite table, is not a power of 2. If you do not quantify the number of partitions by a power of 2, in the worst case the largest partition can be twice the size of the smallest. So for optimal performance, create partitions, or subpartitions per partitions, using a power of two. For example, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, and so on.

The following example creates 4 hashed partitions for the table "Sales" using the column s_productid as the partition key:

```
CREATE TABLE sales
(s_productid NUMBER,
 s_saledate DATE,
 s_custid NUMBER,
 s_totalprice NUMBER)
PARTITION BY HASH(s_productid)
PARTITIONS 4;
```

Specify the partition names only if you want some of the partitions to have different properties than the table. Otherwise, Oracle automatically generates internal names for the partitions. Also, you can use the STORE IN clause to assign partitions to tablespaces in a round-robin manner.

Performance Consideration for Composite Partitioning Composite partitioning offers the benefits of both range and hash partitioning. With composite partitioning, Oracle first partitions by range, and then within each range Oracle creates subpartitions and distributes data within them using a hashing algorithm. Oracle uses the same hashing algorithm to distribute data among the hash subpartitions of composite partitioned tables as it does for hash partitioned tables.

Data placed in composite partitions is logically ordered only in terms of the partition boundaries you use to define the range level partitions. The partitioning of data within each partition has no logical organization beyond the identity of the partition to which the subpartitions belong.

Consequently, tables and local indexes partitioned using the composite method:

- Support historical data at the partition level
- Support the use of subpartitions as units of parallelism for parallel operations such as PDML, for example, space management and backup and recovery
- Are subject to partition pruning and partition-wise joins on the range and hash dimensions

Using Composite Partitioning Use the composite partitioning method for tables and local indexes if:

- Partitions must have a logical meaning to efficiently support historical data
- The contents of a partition may be spread across multiple tablespaces, devices or nodes of an MPP system.
- You need to use both partition pruning and partition-wise joins even when the pruning and join predicates use different columns of the partitioned table
- You want to use a degree of parallelism that is greater than the number of partitions for backup, recovery and parallel operations

When using the composite method, Oracle stores each subpartition on a different segment. Thus, the subpartitions may have properties that are different from the properties of the table or the partition to which the subpartitions belong.

The following SQL example partitions the table “Sales” by range on the column s_saledate to create 4 partitions. This takes advantage of ordering data by a time frame. Then within each range partition, the data is further subdivided into 4 subpartitions by hash on the column s_productid.

```
CREATE TABLE sales(
s_productid NUMBER,
s_saledate DATE,
s_custid NUMBER,
s_totalprice)
PARTITION BY RANGE (s_saledate)
SUBPARTITION BY HASH (s_productid) SUBPARTITIONS 4
```

```
(PARTITION sal94q1 VALUES LESS THAN TO_DATE (01-APR-1994, DD-MON-YYYY),
 PARTITION sal94q2 VALUES LESS THAN TO_DATE (01-JUL-1994, DD-MON-YYYY),
 PARTITION sal94q3 VALUES LESS THAN TO_DATE (01-OCT-1994, DD-MON-YYYY),
 PARTITION sal94q4 VALUES LESS THAN TO_DATE (01-JAN-1995, DD-MON-YYYY));
```

Each hashed subpartition contains sales of a single quarter ordered by product code. The total number of subpartitions is 16.

Partition Pruning

Partition pruning improves query execution by using the cost-based optimizer to analyze FROM and WHERE clauses in SQL statements to eliminate unneeded partitions when building the partition access list. This allows Oracle to only perform operations on partitions relevant to the SQL statement. Oracle can only do this when you use range, equality, and inlist predicates on the range partitioning columns, and equality and inlist predicates on the hash partitioning columns.

Partition pruning can also dramatically reduce the amount of data retrieved from disk and reduce processing time. This results in substantial improvements in query performance and resource utilization. If you partition the index and table on different columns, partition pruning also eliminates index partitions even when the underlying table's partitions cannot be eliminated. Do this by creating a global partitioned index.

On composite partitioned objects, Oracle can prune at both the range partition level and hash subpartition level using the relevant predicates. For example, referring to the table Sales from the previous example, partitioned by range on the column s_saledate and subpartitioned by hash on column s_productid, consider the following SQL statement

```
SELECT * FROM sales
WHERE s_saledate BETWEEN TO_DATE(01-JUL-1994, DD-MON-YYYY) AND
TO_DATE(01-OCT-1994, DD-MON-YYYY) AND s_productid = 1200;
```

Oracle uses the predicate on the partitioning columns to perform partition pruning as follows:

- On the range dimension, Oracle only accesses partitions sal94q2 and sal94q3
- On the hash dimension, Oracle only accesses the third partition, h3, where rows with s_productid equal to 1200 are mapped

Pruning Using DATE Columns

In the previous example the date value was fully specified, 4 digits for year, using the TO_DATE function. While this is the recommended format for specifying date values, the optimizer can prune partitions using the predicates on s_saledate when you use other formats as in the following examples:

```
SELECT * FROM sales
WHERE s_saledate BETWEEN TO_DATE(01-JUL-1994, DD-MON-YY) AND
TO_DATE(01-OCT-1994, DD-MON-YY) AND s_productid = 1200;

SELECT * FROM sales
WHERE s_saledate BETWEEN ’01-JUL-1994’ AND
’01-OCT-1994’ AND s_productid = 1200;
```

However, you will not be able to see which partitions Oracle is accessing as is usually shown on the partition_start and partition_stop columns of the EXPLAIN PLAN command output on the SQL statement. Instead, you will see the keyword ’KEY’ for both columns.

Avoiding I/O Bottlenecks

As mentioned, to avoid I/O bottlenecks, when Oracle is not scanning all partitions because some have been eliminated by pruning, spread each partition over several devices. On MPP systems, spread those devices over multiple nodes.

Partition-wise Joins

Partition-wise joins reduce query response time by minimizing the amount of data exchanged among query servers when joins execute in parallel. This significantly reduces response time and resource utilization, both in terms of CPU and memory. In OPS (Oracle Parallel Server) environments it also avoids or at least limits the data traffic over the interconnect which is the key to achieving good scalability for massive join operations.

There are two variations of partition-wise join, full and partial as discussed under the following headings.

Full Partition-wise Joins

A full partition-wise join divides a large join into smaller joins between a pair of partitions from the two joined tables. To use this feature, you must equi-partition both tables on their join keys. For example, consider a massive join between a sales table and a customer table on the column ’customerid’. The query “find the records of all customers who bought more than 100 articles in Quarter 3 of 1994" is a typical example of a SQL statement performing a massive join. The following is an example of this:

```
SELECT c_customer_name, COUNT(*)
FROM sales, customer
  WHERE s_customerid = c_customerid
 AND s_saledate BETWEEN TO_DATE(01-jul-1994, DD-MON-YYYY) AND
  TO_DATE(01-oct-1994, DD-MON-YYYY)
GROUP BY c_customer_name HAVING
COUNT(*) > 100;
```

This join is a massive join typical in data warehousing environments. The entire customer table is joined with one quarter of the sales data. In large data warehouse applications, it might mean joining millions of rows. The hash method to use in that case is obviously a hash join. But you can reduce the processing time for this hash join even more if both tables are equi-partitioned on the customerid column. This enables a full partition-wise join.

When you execute a full partition-wise join in parallel, the granule of parallelism, as described under "Types of Parallelism" on page 26-36, is a partition. As a result, the degree of parallelism is limited to the number of partitions. For example, you should have at least 16 partitions to set the degree of parallelism of the query to 16.

You can use various partitioning methods to equi-partition both tables on the column customerid with 16 partitions. These methods are described in the following sub-sections.

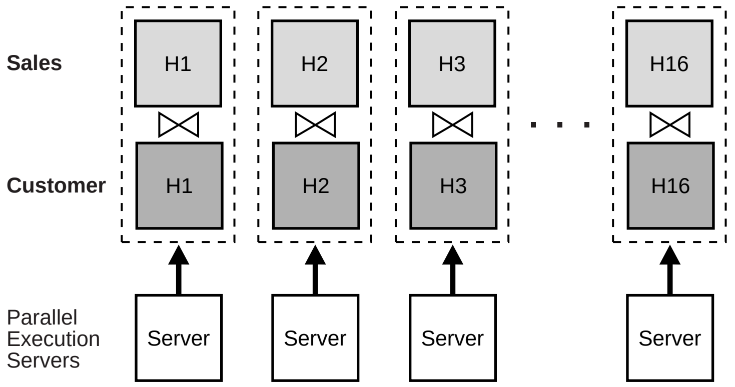
Hash - Hash This is the simplest method: the Customer and Sales tables are both partitioned by hash into 16 partitions, on s_customerid and c_customerid respectively. This partitioning method should enable full partition-wise join when the tables are joined on the customerid column.

In serial, this join is performed between a pair of matching hash partitions at a time: when one partition pair has been joined, the join of another partition pair begins. The join completes when the 16 partition pairs have been processed.

Note: A pair of matching hash partitions is defined as one partition from each table with the same partition number. For example, with full partition wise joins we join partition 0 of sales with partition 0 of customer, partition 1 of sales with partition 1 of customer, and so on.

Parallel execution of a full partition-wise join is a straight-forward parallelization of the serial execution. Instead of joining one partition pair at a time, 16 partition pairs are joined in parallel by the 16 query servers. Figure 26–6 illustrates the parallel execution of a full partition-wise join.

Figure 26–6 Parallel Execution of A Full Partition-wise Join



In Figure 26–6 we assume that the degree of parallelism and the number of partitions are the same, in other words, 16 for both. It is possible to have more partitions than the degree of parallelism to improve load balancing and limit possible skew in the execution. If you have more partitions than query servers, when one query server is done with the join of one pair of partitions, it requests that the query coordinator give it another pair to join. This process repeats until all pairs have been processed. This method allows dynamic load balancing when the number of partition pairs is greater than the degree of parallelism, for example, 64 partitions with a degree of parallelism of 16.

Note: Always use a number of partitions that is a multiple of the degree of parallelism.

In Oracle Parallel Server environments running on shared-nothing platforms or MPPs, partition placements on nodes is critical to achieving good scalability. To avoid remote I/O, both matching partitions should have affinity to the same node. Partition pairs should be spread over all nodes to avoid bottlenecks and to use all CPU resources available on the system.

You can, however, have a node host multiple pairs when there are more pairs than nodes. For example, with an 8-node system and 16 partition pairs, each node should receive 2 pairs.

> **See Also:** For more information on data affinity, please refer to *Oracle8i Parallel Server Concepts and Administration*.

Composite - Hash This method is a variation of the hash / hash method. The sales table is a typical example of a table storing historical data. For all the reasons mentioned under the heading "Performance Considerations for Range Partitioning" on page 26-47, a more logical partitioning method for sales is probably the range method, not the hash method.

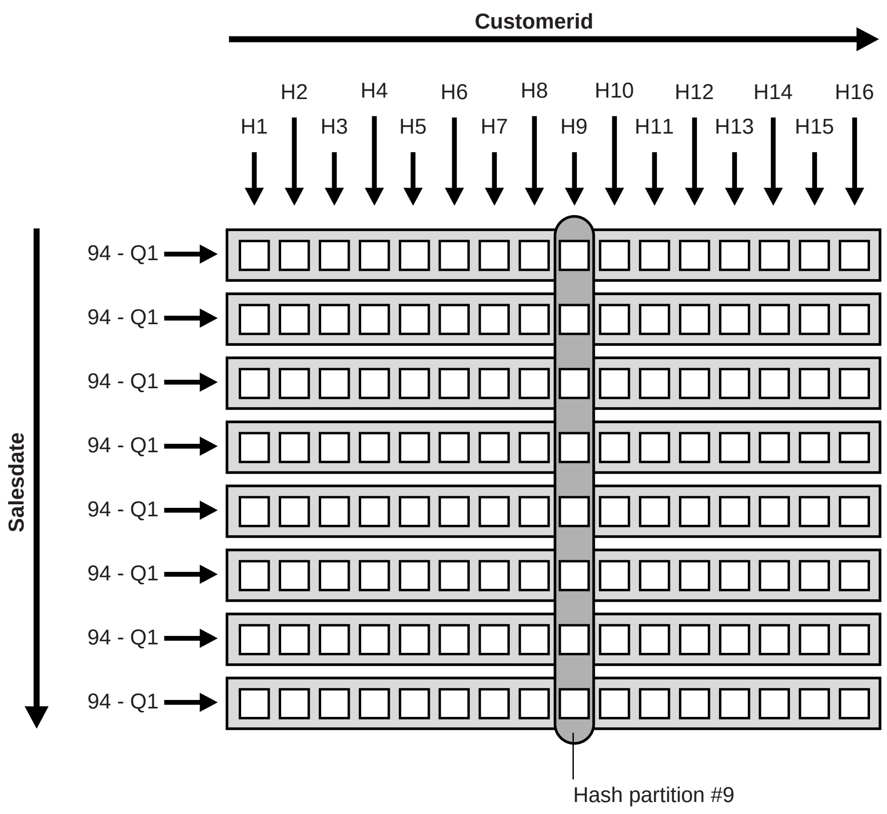
For example, assume you want to partition the Sales table by range on the column s_saledate into 8 partitions. Also assume you have 2 years' of data and each partition represents a quarter. Instead of range partitioning you can use composite to enable a full partition-wise join while preserving the partitioning on s_saledate. Do this by partitioning the Sales table by range on s_saledate and then by subpartitioning each partition by hash on s_customerid using 16 subpartitions per partition, for a total of 128 subpartitions. The customer table can still use hash partitioning with 16 partitions.

With that new partitioning method, a full partition-wise join works similarly to the hash / hash method. The join is still divided into 16 smaller joins between hash partition pairs from both tables. The difference is that now each hash partition in the Sales table is composed of a set of 8 subpartitions, one from each range partition.

Figure 26–7 illustrates how the hash partitions are formed in the Sales table. In Figure 26–7, each cell represents a subpartition. Each row corresponds to one range partition for a total of 8 range partitions; each range partition has 16 subpartitions. Symmetrically, each column on the figure corresponds to one hash partition for a total of 16 hash partitions; each hash partition has 8 subpartitions. Note that hash partitions can be defined only if all partitions have the same number of subpartitions, in this case, 16.

Hash partitions in a composite table are implicit. However, Oracle does not record them in the data dictionary and you cannot manipulate them with DDL commands as you can range partitions.

Figure 26–7 Range and Hash Partitions of A Composite Table



This partitioning method is effective because it allows you to combine pruning (on s_salesdate) with a full partition-wise join (on customerid). In the previous example query, pruning is achieved by only scanning the subpartitions corresponding to Q3 of 1994, in other words, row number 3 on Figure 26–7. Oracle them joins these subpartitions with the customer table using a full partition-wise join.

All characteristics of the hash / hash method also apply to the composite / hash method. In particular for this example, these two points are common to both methods:

- The degree of parallelism for a full partition-wise join cannot exceed 16. This is because even though the Sales table has 128 subpartitions, it has only 16 hash partitions.
- The same rules for data placement on MPP systems apply here. The only difference is that a hash partition is now a collection of subpartitions. You must ensure that all these subpartitions are placed on the same node with the matching hash partition from the other table. For example, in Figure 26–7, you should store hash partition 9 of the Sales table shown by the 8 circled subpartitions, on the same node as hash partition 9 of the Customer table.

Composite - Composite (Hash Dimension) If needed, you can also partition the Customer table by composite. For example, you can partition it by range on a zip code column to enable pruning based on zip code. You should then subpartition it by hash on customerid to enable a partition-wise join on the hash dimension.

Range - Range You can also use partition-wise joins for range partitioning. However, this is more complex to implement because you must know your data's distribution before performing the join. Furthermore, this can lead to data skew during the execution if you do not correctly identify the partition bounds so you have partitions of equal size.

The basic principle for using range / range is the same as for hash / hash: you must equi-partition both tables. This means that the number of partitions must be the same and the partition bounds must be identical. For example, assume that you know in advance that you have 10 million customers, and the values for customerid vary from 1 to 10000000. In other words, you have possibly 10 million different values. To create 16 partitions, you can range partition both tables, Sales on s_customerid and Customer on c_customerid. You should define partition bounds for both tables to generate partitions of the same size. In this example, partition bounds should be defined as 625001, 1250001, 1875001, ..., 10000001, so each partition contains 625000 rows.

Range - Composite, Composite - Composite (Range Dimension) Finally, you can also subpartition one or both tables on another column. Therefore, the range / composite and composite / composite methods on the range dimension are also valid for enabling a full partition-wise join on the range dimension.

Partial Partition-wise Joins

Oracle can only perform partial partition-wise joins in parallel. Unlike full partition-wise joins, partial partition-wise joins require you to partition only one table on the join key, not both. The partitioned table is referred to as the 'reference' table. The other table may or may not be partitioned. Partial partition-wise joins are more common than full partition-wise join because it requires that you only partition one of the joined tables on the join key.

To execute a partial partition-wise join, Oracle dynamically re-partitions the other table based on the partitioning of the reference table. Once the other table is repartitioned, the execution is similar to a full partition-wise join.

The performance advantage that partial partition-wise joins have over conventional parallel joins is that the reference table is not 'moved' during the join operation. Conventional parallel joins require both input tables to be re-distributed on the join key. This re-distribution operation involves exchanging rows between query servers. This is a very CPU-intensive operation and can lead to excessive interconnect traffic in OPS environments. Partitioning large tables on a join key, either a foreign or primary key, prevents this re-distribution every time the table is joined on that key. Of course, if you choose a foreign key to partition the table, which is the most common scenario, select a foreign key that is involved in many queries.

To illustrate partial partition-wise joins, consider the previous Sales/Customer example. Assume that customer is not partitioned or partitioned on a column other than c_customerid. Because Sales is often joined with Customer on customerid and because this join dominates our application workload, partition Sales on s_customerid to enable partial partition-wise join every time Customer and Sales are joined. As in full partition-wise join, we have several alternatives:

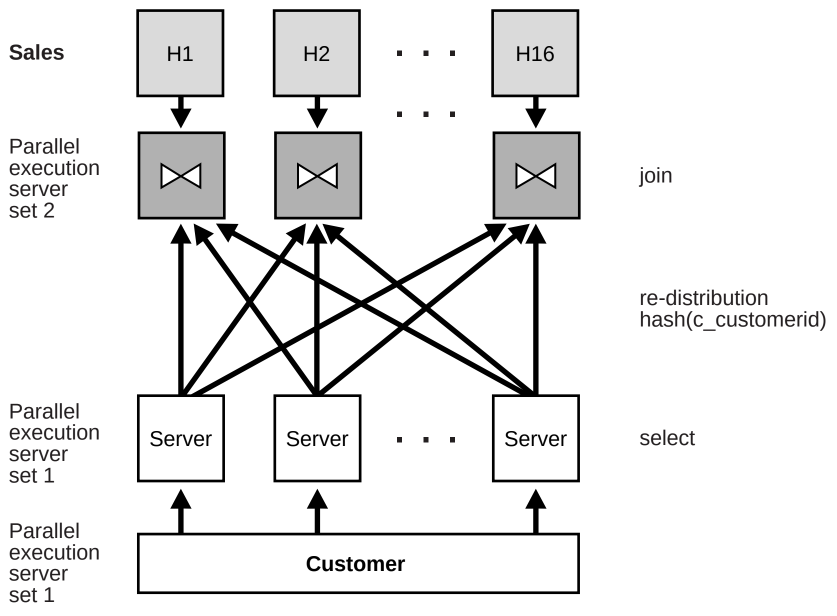
Hash the simplest method to enable a partial partition-wise join is to partition Sales by hash on c_customerid. The number of partitions determines the maximum degree of parallelism because the partition is the smallest granule of parallelism for partial partition-wise join operations.

The parallel execution of a partial partition-wise join is illustrated in Figure 26–8 assuming that both the degree of parallelism and the number of partitions of Sales are 16. The execution involves two sets of query servers: one set, labeled 'set 1' on the figure, scans the customer table in parallel. The granule of parallelism for the scan operation is a range of blocks.

Rows from Customer that are selected by the first set, in this case this is all rows, are re-distributed to the second set of query servers by hashing customerid. For example, all rows in Customer that could have matching rows in partition H1 of Sales are sent to query server 1 in the second set. Rows received by the second set of query servers are joined with the rows from the corresponding partitions in Sales.

For example, query server number 1 in the second set joins all Customer rows that it receives with partition H1 of Sales.

Figure 26–8 Partial Partition-wise Join


Sales
H1
H2
H16
Parallel execution server set 2
join
re-distribution hash(c_customerid)
Parallel execution server set 1
Server
Server
Server
select
Parallel execution server set 1
Customer

Considerations for full partition-wise joins also apply to partial partition-wise joins:

- The degree of parallelism does not need to equal the number of partitions. In Figure 26–8, the query executes with 8 query server sets. In this case, Oracle assigns 2 partitions to each query server of the second set. Again, the number of partitions should always be a multiple of the degree of parallelism.
- In Oracle Parallel Server environments on shared-nothing platforms (MPPs), each hash partition of sales should preferably have affinity to only one node to avoid remote I/Os. Also, spread partitions over all nodes to avoid bottlenecks and use all CPU resources available on the system. It is adequate for a node to host multiple partitions when there are more partitions than nodes.

See Also: For more information on data affinity, please refer to *Oracle8i Parallel Server Concepts and Administration*.

Composite As with full partition-wise joins, the prime partitioning method for the Sales table is to use the range method on column s_salesdate. This is because Sales is a typical example of a table that stores historical data. To enable a partial partition-wise join while preserving this range partitioning, you can subpartition Sales by hash on column s_customerid using 16 subpartitions per partition. Pruning and partial partition-wise joins can be used together if a query joins Customer and Sales and if the query has a selection predicate on s_salesdate.

When Sales is composite, the granule of parallelism for partial-partition wise join is a hash partition and not a subpartition. Refer to Figure 26–7 for the definition of a hash partition in a composite table. Again, the number of hash partitions should be a multiple of the degree of parallelism. Also, on an MPP system, ensure that each hash partition has affinity to a single node. In the previous example, it means that the 8 subpartitions composing a hash partition should have affinity to the same node.

Range finally, you can use range partitioning on s_customerid to enable a partial partition-wise join. This works similarly to the hash method, although it is not recommended. The resulting data distribution could be skewed if the size of the partitions differs. Moreover, this method is more complex to implement because it requires prior knowledge of the values of the partitioning column which is also a join key.

Benefits of Partition-wise Joins

Partition-wise joins offer benefits as described in this section:

- Reduction of Communications Overhead
- Reduction of Memory Requirements

Reduction of Communications Overhead Partition-wise joins reduce communications overhead when they are executed in parallel. This is because in the default case, parallel execution of a join operation by a set of parallel execution servers requires the redistribution of each table on the join column into disjoint subsets of rows. These disjoint subsets of rows are then joined pair-wise by a single parallel execution server.

Oracle can avoid redistributing the partitions since the two tables are already partitioned on the join column. This allows each parallel execution server to join a pair of matching partitions.

This performance enhancement is even more noticeable in OPS configurations with internode parallel execution. This is because partition-wise joins can dramatically reduce interconnect traffic. Using this feature is an almost mandatory optimization measure for large DSS configurations that use OPS.

Currently, most OPS platforms, such as MPP and SMP clusters, provide very limited interconnect bandwidths compared to their processing powers. Ideally, interconnect bandwidth should be comparable to disk bandwidth, but this is seldom the case. As a result, most join operations in OPS experience excessively high interconnect latencies without this optimization.

Reduction of Memory Requirements Partition-wise joins require less memory. In the case of serial joins, the join is performed on a pair of matching partitions at the same time. Thus, if data is evenly distributed across partitions, the memory requirement is divided by the number of partition. In this case there is no skew.

In the parallel case it depends on the number of partition pairs that are joined in parallel. For example, if the degree of parallelism is 20 and the number of partitions is 100, 5 times less memory is required because only 20 joins of two partitions are performed at the same time. The fact that partition-wise joins require less memory has a direct effect on performance. For example, the join does not need to write blocks to disk during the build phase of a hash join.

Performance Considerations for Parallel Partition-wise Joins

The performance improvements from parallel partition-wise joins also come with disadvantages. The cost-based optimizer weighs the advantages and disadvantages when deciding whether to use partition-wise joins.

- In the case of range partitioning, data skew may increase response time if the partitions are of different sizes. This is because some parallel execution servers take longer than others to finish their joins. Oracle recommends the use of hash (sub)partitioning to enable partition-wise joins because hash partitioning limits the risk of skew; assuming that the number of partitions is a power of 2.
- The number of partitions used for partition-wise join should, if possible, be a multiple of the number of query servers. With a degree of parallelism of 16, for example, it is acceptable to have 16, 32 or even 64 partitions (or subpartitions). But Oracle will serially execute the last phase of the join if the degree of parallelism is, for example, 17. This is because in the beginning of the execution,

each parallel execution server works on a different partition pair. At the end of this first phase, only one pair is left. Thus, a single query server joins this remaining pair while all other query servers are idle.

- Sometimes parallel joins can cause remote I/Os. For example, on Oracle Parallel Server environments running on MPP configurations, if a pair of matching partitions is not collocated on the same node, a partition-wise join requires extra inter-node communication due to remote I/Os. This is because Oracle must transfer at least one partition to the node where the join is performed. In this case, it is better to explicitly redistribute the data than to use a partition-wise join.

Phase Three - Creating, Populating, and Refreshing the Database

This section discusses the following topics:

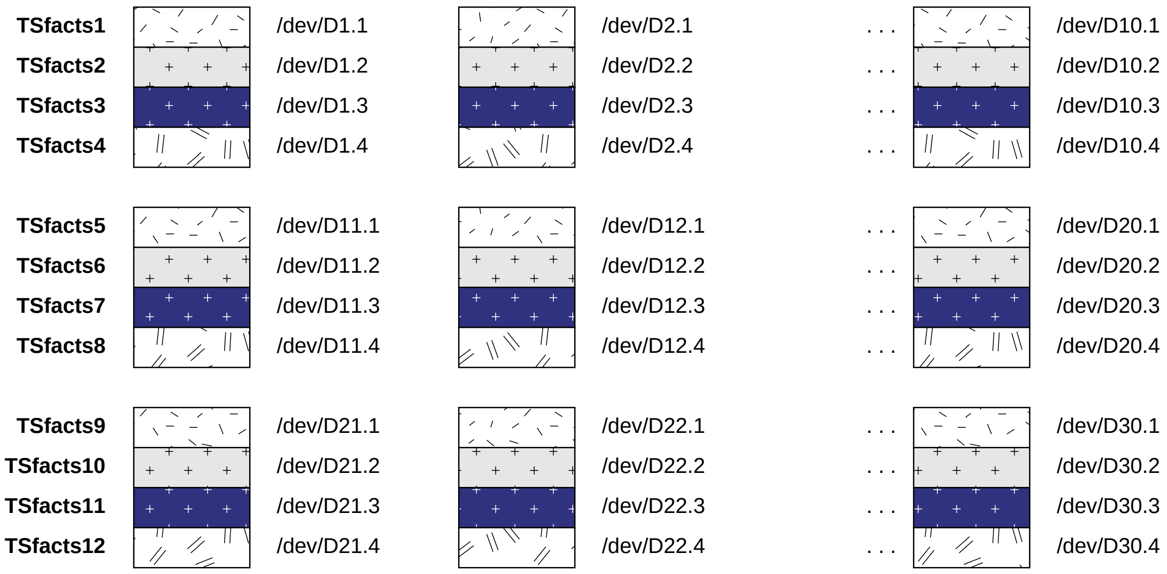
Populating Databases Using Parallel Load

This section presents a case study illustrating how to create, load, index, and analyze a large data warehouse fact table with partitions in a typical star schema. This example uses SQL Loader to explicitly stripe data over 30 disks.

- The example 120 GB table is named FACTS.
- The system is a 10-CPU shared memory computer with more than 100 disk drives.
- Thirty disks (4 GB each) will be used for base table data, 10 disks for index, and 30 disks for temporary space. Additional disks are needed for rollback segments, control files, log files, possible staging area for loader flat files, and so on.
- The FACTS table is partitioned by month into 12 logical partitions. To facilitate backup and recovery, each partition is stored in its own tablespace.
- Each partition is spread evenly over 10 disks, so a scan accessing few partitions or a single partition can proceed with full parallelism. Thus there can be intra-partition parallelism when queries restrict data access by partition pruning.
- Each disk has been further subdivided using an OS utility into 4 OS files with names like **/dev/D1.1, /dev/D1.2, ... , /dev/D30.4.**
- Four tablespaces are allocated on each group of 10 disks. To better balance I/O and parallelize table space creation (because Oracle writes each block in a datafile when it is added to a tablespace), it is best if each of the four tablespaces on each group of 10 disks has its first datafile on a different disk. Thus the first

tablespace has **/dev/D1.1** as its first datafile, the second tablespace has **/dev/D4.2** as its first datafile, and so on, as illustrated in Figure 26–9.

Figure 26–9 Datafile Layout for Parallel Load Example



Step 1: Create the Tablespaces and Add Datafiles in Parallel

Below is the command to create a tablespace named "Tsfacts1". Other tablespaces are created with analogous commands. On a 10-CPU machine, it should be possible to run all 12 CREATE TABLESPACE commands together. Alternatively, it might be better to run them in two batches of 6 (two from each of the three groups of disks).

```
CREATE TABLESPACE Tsfacts1
DATAFILE /dev/D1.1'  SIZE 1024MB REUSE
DATAFILE /dev/D2.1'  SIZE 1024MB REUSE
DATAFILE /dev/D3.1'  SIZE 1024MB REUSE
DATAFILE /dev/D4.1' SIZE 1024MB REUSE
DATAFILE /dev/D5.1'  SIZE 1024MB REUSE
DATAFILE /dev/D6.1'  SIZE 1024MB REUSE
DATAFILE /dev/D7.1'  SIZE 1024MB REUSE
DATAFILE /dev/D8.1' SIZE 1024MB REUSE
DATAFILE /dev/D9.1'  SIZE 1024MB REUSE
DATAFILE /dev/D10.1  SIZE 1024MB REUSE
DEFAULT STORAGE (INITIAL 100MB NEXT 100MB PCTINCREASE 0)
CREATE TABLESPACE Tsfacts2
DATAFILE /dev/D4.2' SIZE 1024MB REUSE
DATAFILE /dev/D5.2'  SIZE 1024MB REUSE
DATAFILE /dev/D6.2'  SIZE 1024MB REUSE
DATAFILE /dev/D7.2'  SIZE 1024MB REUSE
DATAFILE /dev/D8.2' SIZE 1024MB REUSE
DATAFILE /dev/D9.2'  SIZE 1024MB REUSE
DATAFILE /dev/D10.2  SIZE 1024MB REUSE
DATAFILE /dev/D1.2'  SIZE 1024MB REUSE
DATAFILE /dev/D2.2'  SIZE 1024MB REUSE
DATAFILE /dev/D3.2'  SIZE 1024MB REUSE
DEFAULT STORAGE (INITIAL 100MB NEXT 100MB PCTINCREASE 0)
...
CREATE TABLESPACE Tsfacts4
DATAFILE /dev/D10.4'  SIZE 1024MB REUSE
DATAFILE /dev/D1.4'  SIZE 1024MB REUSE
DATAFILE /dev/D2.4'  SIZE 1024MB REUSE
DATAFILE /dev/D3.4  SIZE 1024MB REUSE
DATAFILE /dev/D4.4' SIZE 1024MB REUSE
DATAFILE /dev/D5.4'  SIZE 1024MB REUSE
DATAFILE /dev/D6.4'  SIZE 1024MB REUSE
DATAFILE /dev/D7.4'  SIZE 1024MB REUSE
DATAFILE /dev/D8.4' SIZE 1024MB REUSE
DATAFILE /dev/D9.4'  SIZE 1024MB REUSE
DEFAULT STORAGE (INITIAL 100MB NEXT 100MB PCTINCREASE 0)
...
CREATE TABLESPACE Tsfacts12
DATAFILE /dev/D30.4'  SIZE 1024MB REUSE
DATAFILE /dev/D21.4'  SIZE 1024MB REUSE
DATAFILE /dev/D22.4'  SIZE 1024MB REUSE
DATAFILE /dev/D23.4  SIZE 1024MB REUSE
DATAFILE /dev/D24.4' SIZE 1024MB REUSE
DATAFILE /dev/D25.4'  SIZE 1024MB REUSE
DATAFILE /dev/D26.4'  SIZE 1024MB REUSE
DATAFILE /dev/D27.4'  SIZE 1024MB REUSE
DATAFILE /dev/D28.4' SIZE 1024MB REUSE
DATAFILE /dev/D29.4'  SIZE 1024MB REUSE
DEFAULT STORAGE (INITIAL 100MB NEXT 100MB PCTINCREASE 0)
```

Extent sizes in the STORAGE clause should be multiples of the multiblock read size, where:

blocksize * MULTIBLOCK_READ_COUNT = *multiblock read size*

INITIAL and NEXT should normally be set to the same value. In the case of parallel load, make the extent size large enough to keep the number of extents reasonable, and to avoid excessive overhead and serialization due to bottlenecks in the data dictionary. When PARALLEL=TRUE is used for parallel loader, the INITIAL extent is not used. In this case you can override the INITIAL extent size specified in the tablespace default storage clause with the value specified in the loader control file, for example, 64KB.

Tables or indexes can have an unlimited number of extents provided you have set the COMPATIBLE system parameter to match the current release number, and use the MAXEXTENTS keyword on the CREATE or ALTER command for the tablespace or object. In practice, however, a limit of 10,000 extents per object is reasonable. A table or index has an unlimited number of extents, so set the PERCENT_INCREASE parameter to zero to have extents of equal size.

Note: It is not desirable to allocate extents faster than about 2 or 3 per minute. See "ST (Space Transaction) Enqueue for Sorts and Temporary Data" on page 27-12 for more information. Thus, each process should get an extent that lasts for 3 to 5 minutes. Normally such an extent is at least 50MB for a large object. Too small an extent size incurs significant overhead and this affects performance and scalability of parallel operations. The largest possible extent size for a 4GB disk evenly divided into 4 partitions is 1GB. 100MB extents should perform well. Each partition will have 100 extents. You can then customize the default storage parameters for each object created in the tablespace, if needed.

Step 2: Create the Partitioned Table

We create a partitioned table with 12 partitions, each in its own tablespace. The table contains multiple dimensions and multiple measures. The partitioning column is named "dim_2" and is a date. There are other columns as well.

```
CREATE TABLE fact (dim_1 NUMBER, dim_2 DATE, ...
meas_1 NUMBER, meas_2 NUMBER, ... )
PARALLEL
(PARTITION BY RANGE (dim_2)
PARTITION jan95 VALUES LESS THAN ('02-01-1995') TABLESPACE
TSfacts1
PARTITION feb95 VALUES LESS THAN ('03-01-1995') TABLESPACE
TSfacts2
...
PARTITION dec95 VALUES LESS THAN ('01-01-1996') TABLESPACE
TSfacts12);
```

Step 3: Load the Partitions in Parallel

This section describes four alternative approaches to loading partitions in parallel.

The different approaches to loading help you manage the ramifications of the PARALLEL=TRUE keyword of SQL*Loader that controls whether individual partitions are loaded in parallel. The PARALLEL keyword entails restrictions such as the following:

- Indexes cannot be defined.
- You need to set a small initial extent, because each loader session gets a new extent when it begins, and it doesn't use any existing space associated with the object.
- Space fragmentation issues arise.

However, regardless of the setting of this keyword, if you have one loader process per partition, you are still effectively loading into the table in parallel.

Case 1

In this approach, assume 12 input files are partitioned in the same way as your table. The DBA has 1 input file per partition of the table to be loaded. The DBA starts 12 SQL*Loader sessions concurrently in parallel, entering statements like these:

```
 SQLLDR DATA=jan95.dat DIRECT=TRUE CONTROL=jan95.ctl
SQLLDR DATA=feb95.dat DIRECT=TRUE CONTROL=feb95.ctl
 . . .
 SQLLDR DATA=dec95.dat DIRECT=TRUE CONTROL=dec95.ctl
```

In the example, the keyword PARALLEL=TRUE is *not* set. A separate control file per partition is necessary because the control file must specify the partition into which the loading should be done. It contains a statement such as:

```
LOAD INTO fact partition(jan95)
```

The advantages of this approach are that local indexes are maintained by SQL*Loader. You still get parallel loading, but on a partition level—without the restrictions of the PARALLEL keyword.

A disadvantage is that you must partition the input prior to loading manually.

Case 2

In another common approach, assume an arbitrary number of input files that are not partitioned in the same way as the table. The DBA can adopt a strategy of performing parallel load for each input file individually. Thus if there are 7 input files, the DBA can start 7 SQL*Loader sessions, using statements like the following:

```
SQLLDR DATA=file1.dat DIRECT=TRUE PARALLEL=TRUE
```

Oracle partitions the input data so that it goes into the correct partitions. In this case all the loader sessions can share the same control file, so there is no need to mention it in the statement.

The keyword PARALLEL=TRUE must be used because each of the 7 loader sessions can write into every partition. In case 1, every loader session would write into only 1 partition, because the data was partitioned prior to loading. Hence all the PARALLEL keyword restrictions are in effect.

In this case Oracle attempts to spread the data evenly across all the files in each of the 12 tablespaces—however an even spread of data is not guaranteed. Moreover, there could be I/O contention during the load when the loader processes are attempting to write to the same device simultaneously.

Case 3

In Case 3 (illustrated in the example), the DBA wants precise control over the load. To achieve this, the DBA must partition the input data in the same way as the datafiles are partitioned in Oracle.

This example uses 10 processes loading into 30 disks. To accomplish this, the DBA must split the input into 120 files beforehand. The 10 processes will load the first partition in parallel on the first 10 disks, then the second partition in parallel on the second 10 disks, and so on through the 12th partition. The DBA runs the following commands concurrently as background processes:

```
SQLLDR DATA=jan95.file1.dat DIRECT=TRUE PARALLEL=TRUE FILE=/dev/D1.1
...
SQLLDR DATA=jan95.file10.dat DIRECT=TRUE PARALLEL=TRUE FILE=/dev/D10.1
WAIT;
...
SQLLDR DATA=dec95.file1.dat DIRECT=TRUE PARALLEL=TRUE FILE=/dev/D30.4
```

```
...
SQLLDR DATA=dec95.file10.dat DIRECT=TRUE PARALLEL=TRUE FILE=/dev/D29.4
```

For Oracle Parallel Server, divide the loader session evenly among the nodes. The datafile being read should always reside on the same node as the loader session.

The keyword PARALLEL=TRUE must be used, because multiple loader sessions can write into the same partition. Hence all the restrictions entailed by the PARALLEL keyword are in effect. An advantage of this approach, however, is that it guarantees that all of the data is precisely balanced, exactly reflecting your partitioning.

Note: Although this example shows parallel load used with partitioned tables, the two features can be used independent of one another.

Case 4

For this approach, all partitions must be in the same tablespace. You need to have the same number of input files as datafiles in the tablespace, but you do not need to partition the input the same way in which the table is partitioned.

For example, if all 30 devices were in the same tablespace, then you would arbitrarily partition your input data into 30 files, then start 30 SQL*Loader sessions in parallel. The statement starting up the first session would be similar to the following:

```
SQLLDR DATA=file1.dat DIRECT=TRUE PARALLEL=TRUE FILE=/dev/D1
. . .
SQLLDR DATA=file30.dat DIRECT=TRUE PARALLEL=TRUE FILE=/dev/D30
```

The advantage of this approach is that as in Case 3, you have control over the exact placement of datafiles because you use the FILE keyword. However, you are not required to partition the input data by value because Oracle does that for you.

A disadvantage is that this approach requires all the partitions to be in the same tablespace. This minimizes availability.

Creating Temporary Tablespaces for Parallel Sort and Hash Join

For optimal space management performance use dedicated temporary tablespaces. As with the TSfacts tablespace, we first add a single datafile and later add the remainder in parallel as in this example:

```
CREATE TABLESPACE TStemp TEMPORARY DATAFILE '/dev/D31'
SIZE 4096MB REUSE
DEFAULT STORAGE (INITIAL 10MB NEXT 10MB PCTINCREASE 0);
```

Size of Temporary Extents

Temporary extents are all the same size, because the server ignores the PCTINCREASE and INITIAL settings and only uses the NEXT setting for temporary extents. This helps avoid fragmentation.

As a general rule, temporary extents should be smaller than permanent extents, because there are more demands for temporary space, and parallel processes or other operations running concurrently must share the temporary tablespace. Normally, temporary extents should be in the range of 1MB to 10MB. Once you allocate an extent it is yours for the duration of your operation. If you allocate a large extent but only need to use a small amount of space, the unused space in the extent is tied up.

At the same time, temporary extents should be large enough that processes do not have to spend all their time waiting for space. Temporary tablespaces use less overhead than permanent tablespaces when allocating and freeing a new extent. However, obtaining a new temporary extent still requires the overhead of acquiring a latch and searching through the SGA structures, as well as SGA space consumption for the sort extent pool. Also, if extents are too small, SMON may take a long time dropping old sort segments when new instances start up.

Operating System Striping of Temporary Tablespaces

Operating system striping is an alternative technique you can use with temporary tablespaces. Media recovery, however, offers subtle challenges for large temporary tablespaces. It does not make sense to mirror, use RAID, or back up a temporary tablespace. If you lose a disk in an OS striped temporary space, you will probably have to drop and recreate the tablespace. This could take several hours for our 120GB example. With Oracle striping, simply remove the defective disk from the tablespace. For example, if /dev/D50 fails, enter:

```
ALTER DATABASE DATAFILE '/dev/D50' RESIZE 1K;
ALTER DATABASE DATAFILE '/dev/D50' OFFLINE;
```

Because the dictionary sees the size as 1KB, which is less than the extent size, the corrupt file is not accessed. Eventually, you may wish to recreate the tablespace.

Be sure to make your temporary tablespace available for use:

```
ALTER USER scott TEMPORARY TABLESPACE TStemp;
```

See Also: For MPP systems, see your platform-specific documentation regarding the advisability of disabling disk affinity when using operating system striping.

Creating Indexes in Parallel

Indexes on the fact table can be partitioned or non-partitioned. Local partitioned indexes provide the simplest administration. The only disadvantage is that a search of a local non-prefixed index requires searching all index partitions.

The considerations for creating index tablespaces are similar to those for creating other tablespaces. Operating system striping with a small stripe width is often a good choice, but to simplify administration it is best to use a separate tablespace for each index. If it is a local index you may want to place it into the same tablespace as the partition to which it corresponds. If each partition is striped over a number of disks, the individual index partitions can be rebuilt in parallel for recovery. Alternatively, operating system mirroring can be used. For these reasons the NOLOGGING option of the index creation statement may be attractive for a data warehouse.

Tablespaces for partitioned indexes should be created in parallel in the same manner as tablespaces for partitioned tables.

Partitioned indexes are created in parallel using partition granules, so the maximum DOP possible is the number of granules. Local index creation has less inherent parallelism than global index creation, and so may run faster if a higher DOP is used. The following statement could be used to create a local index on the fact table:

```
CREATE INDEX I on fact(dim_1,dim_2,dim_3) LOCAL
PARTITION jan95 TABLESPACE Tsidx1,
PARTITION feb95 TABLESPACE Tsidx2,
...
PARALLEL(DEGREE 12) NOLOGGING;
```

To backup or restore January data, you only need to manage tablespace Tsidx1.

Note: To maintain optimal performance, analyze your tablespaces frequently so statistics are up-to-date.

See Also: *Oracle8i Concepts* for a discussion of partitioned indexes. For more information on analyzing statistics, please refer to Chapter 7, "Optimizer Modes, Plan Stability, and Hints"

Executing Parallel SQL Statements

After analyzing your tables and indexes, you should see performance improvements that scales linearly with the degree of parallelism used. The following operations should scale:

- Table scans
- NESTED LOOP JOIN
- SORT MERGE JOIN
- HASH JOIN
- "NOT IN"
- GROUP BY
- SELECT DISTINCT
- UNION and UNION ALL
- AGGREGATION
- PL/SQL functions called from SQL
- ORDER BY
- CREATE TABLE AS SELECT
- CREATE INDEX
- REBUILD INDEX
- REBUILD INDEX PARTITION
- MOVE PARTITION
- SPLIT PARTITION
- UPDATE

- DELETE
- INSERT ... SELECT
- ENABLE CONSTRAINT
- STAR TRANSFORMATION

Start with simple parallel operations. Evaluate total I/O throughput with SELECT COUNT(*) FROM facts. Evaluate total CPU power by adding a complex WHERE clause. I/O imbalance may suggest a better physical database layout. After you understand how simple scans work, add aggregation, joins, and other operations that reflect individual aspects of the overall workload. Look for bottlenecks.

Besides query performance you should also monitor parallel load, parallel index creation, and parallel DML, and look for good utilization of I/O and CPU resources.

Using EXPLAIN PLAN to Show Parallel Operations Plans

Use the EXPLAIN PLAN command to see the execution plans for parallel queries. EXPLAIN PLAN output shows optimizer information in the COST, BYTES, and CARDINALITY columns. For more information on using EXPLAIN PLAN, please refer to Chapter 13, "Using EXPLAIN PLAN".

There are several ways to optimize the parallel execution of join statements. You can alter your system's configuration, adjust parameters as discussed earlier in this chapter, or use hints, such as the DISTRIBUTION hint. For more information on hints for parallel operations, please refer to "Hints for Parallel Execution" on page 7-57.

Additional Considerations for Parallel DML

When you want to refresh your data warehouse database using parallel insert, update, or delete on a data warehouse, there are additional issues to consider when designing the physical database. These considerations do not affect parallel execution operations. These issues are:

- Limitation on the Degree of Parallelism
- Using Local and Global Striping
- Increasing INITRANS and MAXTRANS
- Limitation on Available Number of Transaction Free Lists
- Using Multiple Archivers

- [NO]LOGGING Option

PDML and Direct-load Restrictions

A complete listing of PDML and direct-load insert restrictions is found in *Oracle8i Concepts*. If a parallel restriction is violated, the operation is simply performed serially. If a direct-load insert restriction is violated, then the APPEND hint is ignored and a conventional insert is performed. No error message is returned.

Limitation on the Degree of Parallelism

If you are performing parallel insert, update, or delete operations, the DOP is equal to or less than the number of partitions in the table.

See Also: "Phase Three - Creating, Populating, and Refreshing the Database" on page 26-63.

Using Local and Global Striping

Parallel DML works mostly on partitioned tables. It does not use asynchronous I/O and may generate a high number of random I/O requests during index maintenance of parallel UPDATE and DELETE operations. For local index maintenance, local striping is most efficient in reducing I/O contention, because one server process only goes to its own set of disks and disk controllers. Local striping also increases availability in the event of one disk failing.

For global index maintenance, (partitioned or non-partitioned), globally striping the index across many disks and disk controllers is the best way to distribute the number of I/Os.

Increasing INITRANS and MAXTRANS

If you have global indexes, a global index segment and global index blocks are shared by server processes of the same parallel DML statement. Even if the operations are not performed against the same row, the server processes may share the same index blocks. Each server transaction needs one transaction entry in the index block header before it can make changes to a block. Therefore, in the CREATE INDEX or ALTER INDEX statements, you should set INITRANS, the initial number of transactions allocated within each data block, to a large value, such as the maximum DOP against this index. Leave MAXTRANS, the maximum number of concurrent transactions that can update a data block, at its default value, which is the maximum your system can support. This value should not exceed 255.

If you run a DOP of 10 against a table with a global index, all 10 server processes might attempt to change the same global index block. For this reason you must set MAXTRANS to at least 10 so all server processes can make the change at the same time. If MAXTRANS is not large enough, the parallel DML operation fails.

Limitation on Available Number of Transaction Free Lists

Once a segment has been created, the number of process and transaction free lists is fixed and cannot be altered. If you specify a large number of process free lists in the segment header, you may find that this limits the number of transaction free lists that are available. You can abate this limitation the next time you recreate the segment header by decreasing the number of process free lists; this leaves more room for transaction free lists in the segment header.

For UPDATE and DELETE operations, each server process may require its own transaction free list. The parallel DML DOP is thus effectively limited by the smallest number of transaction free lists available on any of the global indexes the DML statement must maintain. For example, if you have two global indexes, one with 50 transaction free lists and one with 30 transaction free lists, the DOP is limited to 30.

The FREELISTS parameter of the STORAGE clause is used to set the number of process free lists. By default, no process free lists are created.

The default number of transaction free lists depends on the block size. For example, if the number of process free lists is not set explicitly, a 4KB block has about 80 transaction free lists by default. The minimum number of transaction free lists is 25.

See Also: *Oracle8i Parallel Server Concepts and Administration* for information about transaction free lists.

Using Multiple Archivers

Parallel DDL and parallel DML operations may generate a large amount of redo logs. A single ARCH process to archive these redo logs might not be able to keep up. To avoid this problem, you can spawn multiple archiver processes. This can be done manually or by using a job queue.

Database Writer Process (DBWn) Workload

Parallel DML operations dirty a large number of data, index, and undo blocks in the buffer cache during a short period of time. If you see a high number of "FREE_BUFFER_WAITS" after querying the V$SYSTEM_EVENT view as in the following syntax:

```
SELECT TOTAL_WAITS FROM V$SYSTEM_EVENT WHERE EVENT = 'FREE BUFFER WAITS';
```

Tune the DBW*n* process(es). If there are no waits for free buffers, the above query does not return any rows.

> **See Also:** "Tuning the Redo Log Buffer" on page 19-6.

[NO]LOGGING Option

The [NO]LOGGING option applies to tables, partitions, tablespaces, and indexes. Virtually no log is generated for certain operations (such as direct-load INSERT) if the NOLOGGING option is used. The NOLOGGING attribute is not specified at the INSERT statement level, but is instead specified when using the ALTER or CREATE command for the table, partition, index, or tablespace.

When a table or index has NOLOGGING set, neither parallel nor serial direct-load INSERT operations generate undo or redo logs. Processes running with the NOLOGGING option set run faster because no redo is generated. However, after a NOLOGGING operation against a table, partition, or index, if a media failure occurs before a backup is taken, then all tables, partitions, and indexes that have been modified may be corrupted.

> **Note:** Direct-load INSERT operations (except for dictionary updates) never generate undo logs. The NOLOGGING attribute does not affect undo, but only redo. To be precise, NOLOGGING allows the direct-load INSERT operation to generate a negligible amount of redo (range-invalidation redo, as opposed to full image redo).

For backward compatibility, [UN]RECOVERABLE is still supported as an alternate keyword with the CREATE TABLE command. This alternate keyword may not be supported, however, in future releases.

At the tablespace level, the logging clause specifies the default logging attribute for all tables, indexes, and partitions created in the tablespace. When an existing tablespace logging attribute is changed by the ALTER TABLESPACE statement, then all tables, indexes, and partitions created *after* the ALTER statement will have the new logging attribute; existing ones will not change their logging attributes. The tablespace level logging attribute can be overridden by the specifications at the table, index, or partition level.

The default logging attribute is LOGGING. However, if you have put the database in NOARCHIVELOG mode, by issuing ALTER DATABASE NOARCHIVELOG, then all operations that can be done without logging will not generate logs, regardless of the specified logging attribute.

See Also: *Oracle8i SQL Reference.*

Phase Four - Monitoring Parallel Execution Performance

Phase Four discusses the following topics for monitoring parallel execution performance:

- Monitoring Parallel Execution Performance with Dynamic Performance Views
- Monitoring Session Statistics
- Monitoring Operating System Statistics

Monitoring Parallel Execution Performance with Dynamic Performance Views

After your system has run for a few days, monitor parallel execution performance statistics to determine whether your parallel processing is optimal. Do this using any of the views discussed in this phase.

View Names in Oracle Parallel Server

In Oracle Parallel Server, global versions of views described in this phase aggregate statistics from multiple instances. The global views have names beginning with "G", such as GV$FILESTAT for V$FILESTAT, and so on.

V$PX_SESSION

The V$PX_SESSION view shows data about query server sessions, groups, sets, and server numbers. Displays real-time data about the processes working on behalf of parallel execution. This table includes information about the requested DOP and actual DOP granted to the operation.

V$PX_SESSTAT

The V$PX_SESSTAT view provides a join of the session information from V$PX_SESSION and the V$SESSTAT table. Thus, all session statistics available to a normal session are available for all sessions performed using parallel execution.

V$PX_PROCESS

The V$PX_PROCESS view contains information about the parallel processes. Includes status, session ID, Process ID and other information.

V$PX_PROCESS_SYSSTAT

The V$PX_PROCESS_SYSSTAT view shows the status of query servers and provides buffer allocation statistics.

V$PQ_SESSTAT

The V$PQ_SESSTAT view shows the status of all current server groups in the system such as data about how queries allocate processes and how the multi-user and load balancing algorithms are affecting the default and hinted values. V$PQ_SESSTAT will be obsolete in a future release.

You may need to adjust some parameter settings to improve performance after reviewing data from these views. In this case, refer to the discussion of "Step Three: Tuning General Parameters" on page 26-9. Query these views periodically to monitor the progress of long-running parallel operations.

Note: For many dynamic performance views, you must set the parameter TIMED_STATISTICS to TRUE in order for Oracle to collect statistics for each view. You can use the ALTER SYSTEM or ALTER SESSION commands to turn TIMED_STATISTICS on and off.

See Also: For more information on collecting statistics with the DBMS_STATS package, refer to Chapter 7, "Optimizer Modes, Plan Stability, and Hints".

V$FILESTAT

The V$FILESTAT view sums read and write requests, the number of blocks, and service times for every datafile in every tablespace. Use V$FILESTAT to diagnose I/O and workload distribution problems.

You can join statistics from V$FILESTAT with statistics in the DBA_DATA_FILES view to group I/O by tablespace or to find the filename for a given file number. Using a ratio analysis, you can determine the percentage of the total tablespace activity used by each file in the tablespace. If you make a practice of putting just one large, heavily accessed object in a tablespace, you can use this technique to identify objects that have a poor physical layout.

You can further diagnose disk space allocation problems using the DBA_EXTENTS view. Ensure that space is allocated evenly from all files in the tablespace. Monitoring V$FILESTAT during a long-running operation and then correlating I/O activity to the EXPLAIN PLAN output is a good way to follow progress.

V$PARAMETER

The V$PARAMETER view lists the name, current value, and default value of all system parameters. In addition, the view shows whether a parameter is a session parameter that you can modify online with an ALTER SYSTEM or ALTER SESSION command.

V$PQ_TQSTAT

The V$PQ_TQSTAT view provides a detailed report of message traffic at the table queue level. V$PQ_TQSTAT data is valid only when queried from a session that is executing parallel SQL statements. A table queue is the pipeline between query server groups or between the parallel coordinator and a query server group or between a query server group and the coordinator. Table queues are represented in EXPLAIN PLAN output by the row labels of PARALLEL_TO_PARALLEL, SERIAL_TO_PARALLEL, or PARALLEL_TO_SERIAL, respectively.

V$PQ_TQSTAT has a row for each query server process that reads from or writes to in each table queue. A table queue connecting 10 consumer processes to 10 producer processes has 20 rows in the view. Sum the bytes column and group by TQ_ID, the table queue identifier, to obtain the total number of bytes sent through each table queue. Compare this with the optimizer estimates; large variations may indicate a need to analyze the data using a larger sample.

Compute the variance of bytes grouped by TQ_ID. Large variances indicate workload imbalances. You should investigate large variances to determine whether the producers start out with unequal distributions of data, or whether the distribution itself is skewed. If the data itself is skewed, this may indicate a low cardinality, or low number of distinct values.

Note: The V$PQ_TQSTAT view will be renamed in a future release to V$PX_TQSTSAT.

V$SESSTAT and V$SYSSTAT

The V$SESSTAT view provides parallel execution statistics for each session. The statistics include total number of queries, DML and DDL statements executed in a session and the total number of intra- and inter-instance messages exchanged during parallel execution during the session.

V$SYSSTAT does the same as V$SESSTAT for the entire system.

See Also: Chapter 16, "Dynamic Performance Views".

Monitoring Session Statistics

These examples use the dynamic performance views just described.

Use V$PX_SESSION to determine the configuration of the server group executing in parallel. In this example, Session ID 9 is the query coordinator, while sessions 7 and 21 are in the first group, first set. Sessions 18 and 20 are in the first group, second set. The requested and granted DOP for this query is 2 as shown by Oracle’s response to the following query:

```
SELECT QCSID, SID, INST_ID "Inst",
SERVER_GROUP "Group", SERVER_SET "Set",
  DEGREE "Degree", REQ_DEGREE "Req Degree"
  FROM GV$PX_SESSION
   ORDER BY QCSID, QCINST_ID, SERVER_GROUP, SERVER_SET;
```

Oracle responds with:

```
QCSID      SID        Inst       Group      Set        Degree     Req Degree
---------- ---------- ---------- ---------- ---------- ---------- ----------
         9          9          1
         9          7          1          1          1          2          2
         9         21          1          1          1          2          2
         9         18          1          1          2          2          2
         9         20          1          1          2          2          2
5 rows selected.
```

Note: For a single instance, select from V$PX_SESSION and do not include the column name "Instance ID".

The processes shown in the output from the previous example using GV$PX_SESSION collaborate to complete the same task. In the next example, we execute a join query to determine the progress of these processes in terms of physical reads. Use this query to track any specific statistic:

```
SELECT QCSID, SID, INST_ID "Inst",
SERVER_GROUP "Group", SERVER_SET "Set" ,
 NAME "Stat Name", VALUE
 FROM GV$PX_SESSTAT A, V$STATNAME B
 WHERE A.STATISTIC# = B.STATISTIC#
 AND NAME LIKE ’PHYSICAL READS’
 AND VALUE > 0
   ORDER BY QCSID, QCINST_ID, SERVER_GROUP, SERVER_SET;
```

Oracle responds with output similar to:

```
QCSID  SID   Inst   Group  Set    Stat Name         VALUE
------ ----- ------ ------ ------ ----------------- ----------
     9     9      1               physical reads    3863
     9     7      1      1      1 physical reads    2
     9    21      1      1      1 physical reads    2
     9    18      1      1      2 physical reads    2
     9    20      1      1      2 physical reads    2
5 rows selected.
```

Use the previous type of query to track statistics in V$STATNAME. Repeat this query as often as required to observe the progress of the query server processes.

The next query uses V$PX_PROCESS to check the status of the query servers.

```
   SELECT * FROM V$PX_PROCESS;
```

Your output should be similar to the following:

```
SERV STATUS    PID    SPID      SID    SERIAL
---- --------- ------ --------- ------ ------
P002 IN USE        16 16955         21   7729
P003 IN USE        17 16957         20   2921
P004 AVAILABLE     18 16959
P005 AVAILABLE     19 16962
P000 IN USE        12 6999          18   4720
P001 IN USE        13 7004           7    234
6 rows selected.
```

See Also: For more details about these views, please refer to the *Oracle8i Reference.*

Monitoring System Statistics

The V$SYSSTAT and V$SESSTAT views contain several statistics for monitoring parallel execution. Use these statistics to track the number of parallel queries, DMLs, DDLs, DFOs, and operations. Each query, DML, or DDL can have multiple parallel operations and multiple DFOs.

In addition, statistics also count the number of query operations for which the DOP was reduced, or downgraded, due to either the adaptive multi-user algorithm or due to the depletion of available parallel execution servers.

Finally, statistics in these views also count the number of messages sent on behalf of parallel execution. The following syntax is an example of how to display these statistics:

```
  SELECT NAME, VALUE FROM GV$SYSSTAT
  WHERE UPPER (NAME) LIKE '%PARALLEL OPERATIONS%'
     OR UPPER (NAME) LIKE '%PARALLELIZED%'
     OR UPPER (NAME) LIKE '%PX%' ;
```

Oracle responds with output similar to:

```
NAME                                                  VALUE
-------------------------------------------------- ----------
queries parallelized                                      347
DML statements parallelized                                 0
DDL statements parallelized                                 0
DFO trees parallelized                                    463
Parallel operations not downgraded                         28
Parallel operations downgraded to serial                   31
Parallel operations downgraded 75 to 99 pct               252
Parallel operations downgraded 50 to 75 pct              128
Parallel operations downgraded 25 to 50 pct                43
Parallel operations downgraded 1 to 25 pct                 12
PX local messages sent                                  74548
PX local messages recv'd                                74128
PX remote messages sent                                     0
PX remote messages recv'd                                   0

14 rows selected.
```

Monitoring Operating System Statistics

There is considerable overlap between information available in Oracle and information available though operating system utilities (such as **sar** and **vmstat** on UNIX-based systems). Operating systems provide performance statistics on I/O, communication, CPU, memory and paging, scheduling, and synchronization primitives. The Oracle V$SESSTAT view provides the major categories of OS statistics as well.

Typically, operating system information about I/O devices and semaphore operations is harder to map back to database objects and operations than is Oracle information. However, some operating systems have good visualization tools and efficient means of collecting the data.

Operating system information about CPU and memory usage is very important for assessing performance. Probably the most important statistic is CPU usage. The goal of *low -level* performance tuning is to become CPU bound on all CPUs. Once this is achieved, you can move up a level and work at the SQL level to find an alternate plan that might be more I/O intensive but uses less CPU.

Operating system memory and paging information is valuable for fine tuning the many system parameters that control how memory is divided among memory-intensive warehouse subsystems like parallel communication, sort, and hash join.

27

Understanding Parallel Execution Performance Issues

This chapter provides a conceptual explanation of parallel execution performance issues and additional performance enhancement techniques. It also summarizes tools and techniques you can use to obtain performance feedback on parallel operations and explains how to resolve performance problems.

See Also: *Oracle8i Concepts*, for basic principles of parallel execution. Also see your operating system-specific Oracle documentation for more information about tuning while using parallel execution. See your operating system-specific Oracle documentation for more information about tuning while using parallel execution.

Understanding Parallel Execution Performance Issues

Formula for Memory, Users, and Parallel Execution Server Processes

A key to the tuning of parallel operations is an understanding of the relationship between memory requirements, the number of users (processes) a system can support, and the maximum number of parallel execution servers. The goal is to obtain dramatic performance enhancements made possible by parallelizing certain operations, and by using hash joins rather than sort merge joins. You must balance this performance goal with the need to support multiple users.

In considering the maximum number of processes a system can support, it is useful to divide the processes into three classes, based on their memory requirements. Table 27–1 defines high, medium, and low memory processes.

Analyze the maximum number of processes that can fit in memory as follows:

Figure 27–1 Formula for Memory/Users/Server Relationship

$$\begin{array}{l} \quad \mathit{sga\_size} \\ + (\#\ \mathit{low\_memory\_processes} * \mathit{low\_memory\_required}) \\ + (\#\ \mathit{medium\_memory\_processes} * \mathit{medium\_memory\_required}) \\ + (\#\ \mathit{high\_memory\_processes} * \mathit{high\_memory\_required}) \\ \hline \quad \mathit{total\ memory\ required} \end{array}$$

Table 27–1 Memory Requirements for Three Classes of Process

Class	Description
Low Memory Processes: 100KB to 1MB	Low Memory Processes include table scans, index lookups, index nested loop joins; single-row aggregates (such as sum or average with no GROUP BYs, or very few groups), and sorts that return only a few rows; and direct loading. This class of Data Warehousing process is similar to OLTP processes in the amount of memory required. Process memory could be as low as a few hundred kilobytes of fixed overhead. You could potentially support thousands of users performing this kind of operation. You can take this requirement even lower by using the multi-threaded server, and support even more users.
Medium Memory Processes: 1MB to 10MB	Medium Memory Processes include large sorts, sort merge join, GROUP BY or ORDER BY operations returning a large number of rows, parallel insert operations that involve index maintenance, and index creation. These processes require the fixed overhead needed by a low memory process, plus one or more sort areas, depending on the operation. For example, a typical sort merge join would sort both its inputs—resulting in two sort areas. GROUP BY or ORDER BY operations with many groups or rows also require sort areas. Look at the EXPLAIN PLAN output for the operation to identify the number and type of joins, and the number and type of sorts. Optimizer statistics in the plan show the size of the operations. When planning joins, remember that you have several choices. The EXPLAIN PLAN statement is described in Chapter 13, "Using EXPLAIN PLAN".
High Memory Processes: 10MB to 100MB	High memory processes include one or more hash joins, or a combination of one or more hash joins with large sorts. These processes require the fixed overhead needed by a low memory process, plus hash area. The hash area size required might range from 8MB to 32MB, and you might need two of them. If you are performing 2 or more serial hash joins, each process uses 2 hash areas. In a parallel operation, each parallel execution server does at most 1 hash join at a time; therefore, you would need 1 hash area size per server. In summary, the amount of hash join memory for an operation equals the DOP multiplied by hash area size, multiplied by the lesser of either 2 or the number of hash joins in the operation.

Note: The process memory requirements of parallel DML (Data Manipulation Language) and parallel DDL (Data Definition Language) operations also depend upon the query portion of the statement.

Setting Buffer Pool Size for Parallel Operations

The formula to calculate the maximum number of processes your system can support (referred to here as *max_processes*) is:

Figure 27–2 Formula for Calculating the Maximum Number of Processes

$$\begin{array}{rl} & \#\ \textit{low\_memory\_processes} \\ + & \#\ \textit{medium\_memory\_processes} \\ + & \#\ \textit{high\_memory\_processes} \\ \hline & \textit{max\_processes} \end{array}$$

In general, if the value for *max_processes* is much larger than the number of users, consider using parallel operations. If *max_processes* is considerably less than the number of users, consider other alternatives, such as those described in "Balancing the Formula" on page 27-5.

With the exception of parallel update and delete, parallel operations do not generally benefit from larger buffer pool sizes. Parallel update and delete benefit from a larger buffer pool when they update indexes. This is because index updates have a random access pattern and I/O activity can be reduced if an entire index or its interior nodes can be kept in the buffer pool. Other parallel operations can benefit only if you can increase the size of the buffer pool and thereby accommodate the inner table or index for a nested loop join.

See Also: "Tuning the Buffer Cache" on page 19-25 for information about setting the buffer pool size.

Balancing the Formula

Use the following techniques to balance the memory / users / server formula given in Figure 27–1:

- Oversubscribing with Attention to Paging
- Reducing the Number of Memory-Intensive Processes
- Decreasing Data Warehousing Memory per Process
- Decreasing Parallelism for Multiple Users

Oversubscribing with Attention to Paging

You can permit the potential workload to exceed the limits recommended in the formula. Total memory required, minus the SGA size, can be multiplied by a factor of 1.2, to allow for 20% oversubscription. Thus, if you have 1GB of memory, you might be able to support 1.2GB of demand: the other 20% could be handled by the paging system.

You must, however, verify that a particular degree of oversubscription is viable on your system. Do this by monitoring the paging rate and making sure you are not spending more than a very small percent of the time waiting for the paging subsystem. Your system may perform acceptably even if oversubscribed by 60%, if on average not all of the processes are performing hash joins concurrently. Users might then try to use more than the available memory, so you must continually monitor paging activity in such a situation. If paging dramatically increases, consider other alternatives.

On average, no more than 5% of the time should be spent simply waiting in the operating system on page faults. More than 5% wait time indicates your paging subsystem is I / O bound. Use your operating system monitor to check wait time: The sum of time waiting and time running equals 100%. If your total system load is close to 100% of your CPU, your system is not spending a lot of time waiting. If you are waiting, it is not be due to paging.

If wait time for paging devices exceeds 5%, you must most likely reduce memory requirements in one of these ways:

- Reducing the memory required for each class of process
- Reducing the number of processes in memory-intensive classes
- Adding memory

If the wait time indicates an I/O bottleneck in the paging subsystem, you could resolve this by striping.

Reducing the Number of Memory-Intensive Processes

This section describes two things you can do to reduce the number of memory-intensive processes:

- Adjusting the Degree of Parallelism
- Scheduling Parallel Jobs

Adjusting the Degree of Parallelism. You can adjust not only the number of operations that run in parallel, but also the DOP (degree of parallelism) with which operations run. To do this, issue an ALTER TABLE statement with a PARALLEL clause, or use a hint.

> **See Also:** For more information on parallel execution, refer to Chapter 26, "Tuning Parallel Execution". For more information about the ALTER TABLE statement, refer to the *Oracle8i SQL Reference.*

You can limit the parallel pool by reducing the value of PARALLEL_MAX_SERVERS. Doing so places a system-level limit on the total amount of parallelism. It also makes your system easier to administer. More processes are then forced to run in serial mode.

If you enable the parallel adaptive multi-user feature by setting the PARALLEL_ADATIVE_MULTI_USER parameter to TRUE, Oracle controls adjusts DOP based on user load.

> **See Also:** For more information on the parallel adaptive multi-user feature, please refer to "Degree of Parallelism and Adaptive Multi-User and How They Interact" on page 26-7.

Scheduling Parallel Jobs Queueing jobs is another way to reduce the number of processes but not reduce parallelism. Rather than reducing parallelism for all operations, you may be able to schedule large parallel batch jobs to run with full parallelism one at a time, rather than concurrently. Queries at the head of the queue would have a fast response time, those at the end of the queue would have a slow response time. However, this method entails a certain amount of administrative overhead.

Decreasing Data Warehousing Memory per Process

The following discussion focuses upon the relationship of HASH_AREA_SIZE to memory, but all the same considerations apply to SORT_AREA_SIZE. The lower bound of SORT_AREA_SIZE, however, is not as critical as the 8MB recommended minimum HASH_AREA_SIZE.

If every operation performs a hash join and a sort, the high memory requirement limits the number of processes you can have. To allow more users to run concurrently you may need to reduce the data warehouse's process memory.

Moving Processes from High to Medium Memory Requirements You can move a process from the high-memory to the medium-memory class by reducing the value for HASH_AREA_SIZE. With the same amount of memory, Oracle always processes hash joins faster than sort merge joins. Therefore, Oracle does not recommend that you make your hash area smaller than your sort area.

Moving Processes from High or Medium Memory Requirements to Low Memory Requirements If you need to support thousands of users, create access paths so operations do not access data unnecessarily. To do this, perform one or more of the following:

- Decrease the demand for index joins by creating indexes and/or summary tables.
- Decrease the demand for GROUP BY sorting by creating summary tables and encouraging users and applications to reference summaries and materialized views rather than detailed data.
- Decrease the demand for ORDER BY sorts by creating indexes on frequently sorted columns.

See Also: For more information about summary tables, please refer to Section VI, "Materialized Views".

Decreasing Parallelism for Multiple Users

The easiest way to decrease parallelism for multiple users is to enable the parallel adaptive multi-user feature as described under the heading "Degree of Parallelism and Adaptive Multi-User and How They Interact" on page 26-7.

If you decide to control this manually, however, there is a trade-off between parallelism for fast single-user response time and efficient use of resources for multiple users. For example, a system with 2GB of memory and a HASH_AREA_SIZE of 32MB can support about 60 parallel execution servers. A 10 CPU machine can support up to 3 concurrent parallel operations (2 * 10 * 3 = 60). To support 12 concurrent parallel operations, you could override the default parallelism (reduce it), decrease HASH_AREA_SIZE, buy more memory, or you could use some combination of these three strategies. Thus you could ALTER TABLE *t* PARALLEL (DOP = 5) for all parallel tables *t*, set HASH_AREA_SIZE to 16MB, and increase PARALLEL_MAX_SERVERS to 120. By reducing the memory of each parallel server by a factor of 2, and reducing the parallelism of a single operation by a factor of 2, the system can accommodate 2 * 2 = 4 times more concurrent parallel operations.

The penalty for using such an approach is that when a single operation happens to be running, the system uses just half the CPU resource of the 10 CPU machine. The other half is idle until another operation is started.

To determine whether your system is being fully utilized, use one of the graphical system monitors available on most operating systems. These monitors often give you a better idea of CPU utilization and system performance than monitoring the execution time of an operation. Consult your operating system documentation to determine whether your system supports graphical system monitors.

Examples: Balancing Memory, Users, and Parallel Execution Servers

The examples in this section show how to evaluate the relationship between memory, users, and parallel execution servers, and balance the formula given in Figure 27–1. They show concretely how you might adjust your system workload to accommodate the necessary number of processes and users.

Example 1

Assume your system has 1GB of memory, the DOP is 10, and that your users perform 2 hash joins with 3 or more tables. If you need 300MB for the SGA, that leaves 700MB to accommodate processes. If you allow a generous hash area size, such as 32MB, then your system can support:

Figure 27–3   Formula for Balancing Memory, Users, and Processes

1 parallel operation (32MB * 10 * 2 = 640MB)
1 serial operation (32MB * 2 = 64MB)

This makes a total of 704MB. In this case, the memory is not significantly oversubscribed.

Remember that every parallel, hash, or sort merge join operation takes a number of parallel execution servers equal to twice the DOP, utilizing 2 server sets, and often each individual process of a parallel operation uses a significant amount of memory. Thus you can support many more users by running their processes serially, or by using less parallelism to run their processes.

To service more users, you can reduce hash area size to 2MB. This configuration can support 17 parallel operations, or 170 serial operations, but response times may be significantly higher than if you were using hash joins.

The trade-off in this example reveals that by reducing memory per process by a factor of 16, you can increase the number of concurrent users by a factor of 16. Thus the amount of physical memory on the machine imposes another limit on the total number of parallel operations you can run involving hash joins and sorts.

Example 2

In a mixed workload example, consider a user population with diverse needs, as described in Table 27–2. In this situation, you would have to selectively allocate resources. You could not allow everyone to run hash joins—even though they outperform sort merge joins—because you do not have adequate memory to support workload level.

You might consider it safe to oversubscribe by 50%, because of the infrequent batch jobs that run during the day: 700MB * 1.5 = 1.05GB. This gives you enough virtual memory for the total workload.

Table 27–2 How to Accommodate a Mixed Workload

User Needs	How to Accommodate
DBA: runs nightly batch jobs, and occasional batch jobs during the day. These might be parallel operations that perform hash joins and thus use a lot of memory.	You might take 20 parallel execution servers, and set HASH_AREA_SIZE to a mid-range value, perhaps 20MB, for a single powerful batch job in the high memory class. This might be a large GROUP BY operation with a join to produce a summary of data. Twenty servers multiplied by 20MB equals 400MB of memory.
Analysts: interactive users who extract data for their spreadsheets.	You might plan for 10 analysts running serial operations that use complex hash joins accessing a large amount of data. You would not allow them to perform parallel operations because of memory requirements. Ten such serial processes at 40MB each equals 400MB of memory.
Users: Several hundred users performing simple lookups of individual customer accounts, and making reports on already joined, partially summarized data.	To support hundreds of users performing low memory processes at about 0.5MB each, you might reserve 200MB.

Example 3

Suppose your system has 2GB of memory and you have 200 query server processes and 100 users doing performing heavy data warehousing operations involving hash joins. You decide not to consider tasks such as index retrievals and small sorts. Instead, you concentrate on the high memory processes. You might have 300 processes, of which 200 must come from the parallel pool and 100 are single threaded. One quarter of the total 2GB of memory might be used by the SGA, leaving 1.5GB of memory to handle all the processes. You could apply the formula considering only the high memory requirements, including a factor of 20% oversubscription:

Figure 27–4 Formula for Memory/User/Server Relationship: High-Memory Processes

$$high\_memory\_req'd = \frac{total\_memory}{\#\_high\text{-}memory\_processes} * 1.2 = \frac{1.5\text{GB} * 1.2}{300} = \frac{1.8\text{GB}}{300}$$

Here, 5MB = 1.8GB / 300. Less than 5MB of hash area would be available for each process, whereas 8MB is the recommended minimum. If you must have 300 processes, you may need to reduce hash area size to change them from the highly memory-intensive class to the moderately memory-intensive class. Then they may fit within your system's constraints.

Example 4

Consider a system with 2GB of memory and 10 users who want to run intensive data warehousing parallel operations concurrently and still have good performance. If you choose a DOP of 10, then the 10 users will require 200 processes. (Processes running large joins need twice the number of parallel execution servers as the DOP, so you would set PARALLEL_MAX_SERVERS to 10 * 10 * 2.) In this example each process would get 1.8GB / 200—or about 9MB of hash area—which should be adequate.

With only 5 users doing large hash joins, each process would get over 16MB of hash area, which would be fine. But if you want 32MB available for lots of hash joins, the system could only support 2 or 3 users. By contrast, if users are just computing aggregates the system needs adequate sort area size—and can have many more users.

Example 5

If a system with 2GB of memory needs to support 1000 users, all of them running large queries, you must evaluate the situation carefully. Here, the per-user memory budget is only 1.8MB (that is, 1.8GB divided by 1,000). Since this figure is at the low end of the medium memory process class, you must rule out parallel operations, which use even more resources. You must also rule out large hash joins. Each sequential process could require up to 2 hash areas plus the sort area, so you would have to set HASH_AREA_SIZE to the same value as SORT_AREA_SIZE, which would be 600KB(1.8MB / 3). Such a small hash area size is likely to be ineffective.

Given the organization's resources and business needs, is it reasonable for you to upgrade your system's memory? If memory upgrade is not an option, then you must change your expectations. To adjust the balance you might:

- Accept the fact that the system will actually support a limited number of users executing large hash joins.
- Give the users access to summary tables, rather than to the whole database.
- Classify users into different groups, and give some groups more memory than others. Instead of all users doing sorts with a small sort area, you could have a few users doing high-memory hash joins, while most users use summary tables or do low-memory index joins. (You could accomplish this by forcing users in

each group to use hints in their queries such that operations are performed in a particular way.)

Parallel Execution Space Management Issues

This section describes space management issues that occur when using parallel execution. These issues are:

- ST (Space Transaction) Enqueue for Sorts and Temporary Data
- External Fragmentation

These problems become particularly important for parallel operations in an OPS (Oracle Parallel Server) environment; the more nodes that are involved, the more tuning becomes critical.

If you can implement locally-managed tablespaces, you can avoid these issues altogether.

Note: For more information about locally-managed tablespaces, please refer to the *Oracle8i Administrator's Guide*.

ST (Space Transaction) Enqueue for Sorts and Temporary Data

Every space management transaction in the database (such as creation of temporary segments in PARALLEL CREATE TABLE, or parallel direct-load inserts of non-partitioned tables) is controlled by a single ST enqueue. A high transaction rate, for example, more than 2 or 3 transactions per minute, on ST enqueues may result in poor scalability on OPS with many nodes, or a timeout waiting for space management resources. Use the V$ROWCACHE and V$LIBRARYCACHE views to locate this type of contention.

Try to minimize the number of space management transactions, in particular:

- The number of sort space management transactions
- The creation and removal of objects
- Transactions caused by fragmentation in a tablespace

Use dedicated temporary tablespaces to optimize space management for sorts. This is particularly beneficial on OPS. You can monitor this using V$SORT_SEGMENT.

Set INITIAL and NEXT extent size to a value in the range of 1MB to 10MB. Processes may use temporary space at a rate of up to 1MB per second. Do not accept the default value of 40KB for next extent size, because this will result in many requests for space per second.

External Fragmentation

External fragmentation is a concern for parallel load, direct-load insert, and PARALLEL CREATE TABLE ... AS SELECT. Memory tends to become fragmented as extents are allocated and data is inserted and deleted. This may result in a fair amount of free space that is unusable because it consists of small, non-contiguous chunks of memory.

To reduce external fragmentation on partitioned tables, set all extents to the same size. Set the value for NEXT equal to the value for INITIAL and set PERCENT_INCREASE to zero. The system can handle this well with a few thousand extents per object. Therefore, set MAXEXTENTS to, for example, 1,000 to 3,000; never attempt to use a value for MAXEXTENS in excess of 10,000. For tables that are not partitioned, the initial extent should be small.

Tuning Parallel Execution on Oracle Parallel Server

This section describe several aspects of parallel execution for OPS.

Lock Allocation

This section provides parallel execution tuning guidelines for optimal lock management on OPS.

To optimize parallel execution on OPS, you need to correctly set GC_FILES_TO_LOCKS. On OPS, a certain number of parallel cache management (PCM) locks are assigned to each data file. Data block address (DBA) locking in its default behavior assigns one lock to each block. During a full table scan a PCM lock must then be acquired for each block read into the scan. To speed up full table scans, you have three possibilities:

- For data files containing truly read-only data, set the tablespace to read only. Then PCM locking does not occur.
- Alternatively, for data that is mostly read-only, assign very few hashed PCM locks (for example, 2 shared locks) to each data file. Then these are the only locks you have to acquire when you read the data.
- If you want DBA or fine-grain locking, group together the blocks controlled by each lock, using the ! option. This has advantages over default DBA locking because with the default, you would need to acquire one million locks in order

to read one million blocks. When you group the blocks you reduce the number of locks allocated by the grouping factor. Thus a grouping of !10 would mean that you would only have to acquire one tenth as many PCM locks as with the default. Performance improves due to the dramatically reduced amount of lock allocation. As a rule of thumb, performance with a grouping of !10 is comparable to the speed of hashed locking.

To speed up parallel DML operations, consider using hashed locking or a high grouping factor rather than database address locking. A parallel execution server works on non-overlapping partitions; it is recommended that partitions not share files. You can thus reduce the number of lock operations by having only 1 hashed lock per file. Because the parallel execution server only works on non-overlapping files, there are no lock pings.

The following guidelines effect memory usage, and thus indirectly affect performance:

- Never allocate PCM locks for datafiles of temporary tablespaces.
- Never allocate PCM locks for datafiles that contain only rollback segments. These are protected by GC_ROLLBACK_LOCKS and GC_ROLLBACK_SEGMENTS.
- Allocate specific PCM locks for the SYSTEM tablespace. This practice ensures that data dictionary activity such as space management never interferes with the data tablespaces at a cache management level (error 1575).

  For example, on a read-only database with a data warehousing application's query-only workload, you might create 500 PCM locks on the SYSTEM tablespace in file 1, then create 50 more locks to be shared for all the data in the other files. Space management work will never interfere with the rest of the database.

See Also: *Oracle8i Parallel Server Concepts and Administration* for a thorough discussion of PCM locks and locking parameters.

Load Balancing for Multiple Concurrent Parallel Operations

Load balancing distributes query server processes to spread CPU and memory use evenly among nodes. It also minimizes communication and remote I/O among nodes. Oracle does this by allocating servers to the nodes that are running the fewest number of processes.

The load balancing algorithm attempts to maintain an even load across all nodes. For example, if a DOP of 8 is requested on an 8-node MPP (Massively Parallel

Processing) system with 1 CPU per node, the algorithm places 2 servers on each node.

If the entire query server group fits on one node, the load balancing algorithm places all the processes on a single node to avoid communications overhead. For example, if a DOP of 8 is requested on a 2-node cluster with 16 CPUs per node, the algorithm places all 16 query server processes on one node.

Using Parallel Instance Groups

A user or the DBA can control which instances allocate query server processes by using "Instance Group" functionality. To use this feature, you must first assign each active instance to at least one or more instance groups. Then you can dynamically control which instances spawn parallel processes by activating a particular group of instances.

Establish instance group membership on an instance-by-instance basis by setting the initialization parameter INSTANCE_GROUPS to a name representing one or more instance groups. For example, on a 32-node MPP system owned by both a Marketing and a Sales organization, you could assign half the nodes to one organization and the other half to the other organization using instance group names. To do this, assign nodes 1-16 to the Marketing organization using the following parameter syntax in each INIT.ORA file:

```
INSTANCE_GROUPS=marketing
```

Then assign nodes 17-32 to Sales using this syntax in the remaining INIT.ORA files:

```
INSTANCE_GROUPS=sales
```

Then a user or the DBA can activate the nodes owned by Sales to spawn query server process by entering the following:

```
ALTER SESSION SET PARALLEL_INSTANCE_GROUP = 'sales';
```

In response, Oracle allocates query server processes to nodes 17-32. The default value for PARALLEL_INSTANCE_GROUP is all active instances.

Note: As mentioned, an instance can belong to one or more groups. You can enter multiple instance group names with the INSTANCE_GROUP parameter using a comma as a separator.

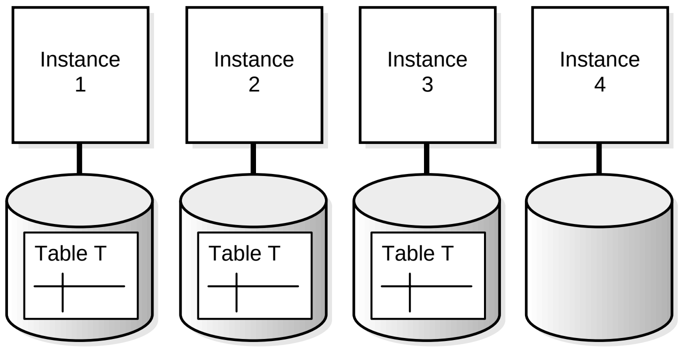
Disk Affinity

Some OPS platforms use disk affinity. Without disk affinity, Oracle tries to balance the allocation evenly across instances; with disk affinity, Oracle tries to allocate parallel execution servers for parallel table scans on the instances that are closest to the requested data. Disk affinity minimizes data shipping and internode communication on a shared nothing architecture. Disk affinity can thus significantly increase parallel operation throughput and decrease response time.

Disk affinity is used for parallel table scans, parallel temporary tablespace allocation, parallel DML, and parallel index scan. It is not used for parallel table creation or parallel index creation. Access to temporary tablespaces preferentially uses local datafiles. It guarantees optimal space management extent allocation. Disks striped by the operating system are treated by disk affinity as a single unit.

In the following example of disk affinity, table T is distributed across 3 nodes, and a full table scan on table T is being performed.

Figure 27–5 Disk Affinity Example



- If a query requires 2 instances, then two instances from the set 1, 2, and 3 are used.
- If a query requires 3 instances, then instances 1, 2, and 3 are used.
- If a query requires 4 instances, then all four instances are used.
- If there are two concurrent operations against table T, each requiring 3 instances (and enough processes are available on the instances for both operations), then both operations use instances 1, 2, and 3. Instance 4 is not used. In contrast, without disk affinity instance 4 is used.

See Also: *Oracle8i Parallel Server Concepts and Administration* for more information on instance affinity.

Parallel Execution Tuning Tips

This section describes performance techniques for parallel operations.

- Overriding the Default Degree of Parallelism
- Rewriting SQL Statements
- Creating and Populating Tables in Parallel
- Creating Indexes in Parallel
- Diagnosing Problems
- Refreshing Tables in Parallel
- Using Hints with Cost Based Optimization

Overriding the Default Degree of Parallelism

The default DOP is appropriate for reducing response time while guaranteeing use of CPU and I/O resources for any parallel operations. If an operation is I/O bound, consider increasing the default DOP. If it is memory bound, or several concurrent parallel operations are running, you might want to decrease the default DOP.

Oracle uses the default DOP for tables that have PARALLEL attributed to them in the data dictionary, or when the PARALLEL hint is specified. If a table does not have parallelism attributed to it, or has NOPARALLEL (the default) attributed to it, then that table is never scanned in parallel—regardless of the default DOP that would be indicated by the number of CPUs, instances, and devices storing that table.

Use the following guidelines when adjusting the DOP:

- You can modify the default DOP by changing the value for the PARALLEL_THREADS_PER_CPU parameter.
- You can adjust the DOP either by using ALTER TABLE or by using hints.
- To increase the number of concurrent parallel operations, reduce the DOP, or set the parameter PARALLEL_ADAPTIVE_MULTI_USER to TRUE.

- For I/O-bound parallel operations, first spread the data over more disks than there are CPUs. Then, increase parallelism in stages. Stop when the query becomes CPU bound.

  For example, assume a parallel indexed nested loop join is I/O bound performing the index lookups, with *#CPU*s=10 and *#disks*=36. The default DOP is 10, and this is I/O bound. You could first try a DOP of 12. If the application is still I/O bound, try a DOP of 24; if still I/O bound, try 36.

Rewriting SQL Statements

The most important issue for parallel execution is ensuring that all parts of the query plan that process a substantial amount of data execute in parallel. Use EXPLAIN PLAN to verify that all plan steps have an OTHER_TAG of PARALLEL_TO_PARALLEL, PARALLEL_TO_SERIAL, PARALLEL_COMBINED_WITH_PARENT, or PARALLEL_COMBINED_WITH_CHILD. Any other keyword (or null) indicates serial execution, and a possible bottleneck.

By making the following changes you can increase the optimizer's ability to generate parallel plans:

- Convert subqueries, especially correlated subqueries, into joins. Oracle can parallelize joins more efficiently than subqueries. This also applies to updates.
- Use a PL/SQL function in the WHERE clause of the main query, instead of a correlated subquery.
- Rewrite queries with distinct aggregates as nested queries. For example, rewrite

  ```
  SELECT COUNT(DISTINCT C) FROM T;
  ```

  To:

  ```
  SELECT COUNT(*)FROM (SELECT DISTINCT C FROM T);
  ```

See Also: "Updating the Table" on page 27-25.

Creating and Populating Tables in Parallel

Oracle cannot return results to a user process in parallel. If a query returns a large number of rows, execution of the query may indeed be faster; however, the user process can only receive the rows serially. To optimize parallel execution performance with queries that retrieve large result sets, use PARALLEL CREATE

TABLE ... AS SELECT or direct-load insert to store the result set in the database. At a later time, users can view the result set serially.

Note: Parallelism of the SELECT does not influence the CREATE statement. If the CREATE is parallel, however, the optimizer tries to make the SELECT run in parallel also.

When combined with the NOLOGGING option, the parallel version of CREATE TABLE ... AS SELECT provides a very efficient intermediate table facility.

For example:

```
CREATE TABLE summary PARALLEL NOLOGGING
  AS SELECT dim_1, dim_2 ..., SUM (meas_1) FROM facts
  GROUP BY dim_1, dim_2;
```

These tables can also be incrementally loaded with parallel insert. You can take advantage of intermediate tables using the following techniques:

- Common subqueries can be computed once and referenced many times. This may be much more efficient than referencing a complex view many times.
- Decompose complex queries into simpler steps in order to provide application-level checkpoint / restart. For example, a complex multi-table join on a database 1 terabyte in size could run for dozens of hours. A crash during this query would mean starting over from the beginning. Using CREATE TABLE ... AS SELECT and / or PARALLEL INSERT AS SELECT, you can rewrite the query as a sequence of simpler queries that run for a few hours each. If a system failure occurs, the query can be restarted from the last completed step.
- Materialize a Cartesian product. This may allow queries against star schemas to execute in parallel. It may also increase scalability of parallel hash joins by increasing the number of distinct values in the join column.

  Consider a large table of retail sales data that is joined to region and to department lookup tables. There are 5 regions and 25 departments. If the huge table is joined to regions using parallel hash partitioning, the maximum speedup is 5. Similarly, if the huge table is joined to departments, the maximum speedup is 25. But if a temporary table containing the Cartesian product of regions and departments is joined with the huge table, the maximum speedup is 125.
- Efficiently implement manual parallel deletes by creating a new table that omits the unwanted rows from the original table, and then dropping the original

table. Alternatively, you can use the convenient parallel delete feature, which can directly delete rows from the original table.

- Create summary tables for efficient multidimensional drill-down analysis. For example, a summary table might store the sum of revenue grouped by month, brand, region, and salesperson.
- Reorganize tables, eliminating chained rows, compressing free space, and so on, by copying the old table to a new table. This is much faster than export/import and easier than reloading.

Note: Be sure to use the ANALYZE statement on newly created tables. Also consider creating indexes. To avoid I/O bottlenecks, specify a tablespace with at least as many devices as CPUs. To avoid fragmentation in allocating space, the number of files in a tablespace should be a multiple of the number of CPUs.

Creating Indexes in Parallel

Multiple processes can work together simultaneously to create an index. By dividing the work necessary to create an index among multiple server processes, the Oracle Server can create the index more quickly than if a single server process created the index sequentially.

Parallel index creation works in much the same way as a table scan with an ORDER BY clause. The table is randomly sampled and a set of index keys is found that equally divides the index into the same number of pieces as the DOP. A first set of query processes scans the table, extracts key, ROWID pairs, and sends each pair to a process in a second set of query processes based on key. Each process in the second set sorts the keys and builds an index in the usual fashion. After all index pieces are built, the parallel coordinator simply concatenates the pieces (which are ordered) to form the final index.

Parallel local index creation uses a single server set. Each server process in the set is assigned a table partition to scan, and to build an index partition for. Because half as many server processes are used for a given DOP, parallel local index creation can be run with a higher DOP.

You can optionally specify that no redo and undo logging should occur during index creation. This can significantly improve performance, but temporarily renders the index unrecoverable. Recoverability is restored after the new index is backed up. If your application can tolerate this window where recovery of the index requires it to be re-created, then you should consider using the NOLOGGING option.

The PARALLEL clause in the CREATE INDEX statement is the only way in which you can specify the DOP for creating the index. If the DOP is not specified in the parallel clause of CREATE INDEX, then the number of CPUs is used as the DOP. If there is no parallel clause, index creation is done serially.

Note: When creating an index in parallel, the STORAGE clause refers to the storage of each of the subindexes created by the query server processes. Therefore, an index created with an INITIAL of 5MB and a DOP of 12 consumes at least 60MB of storage during index creation because each process starts with an extent of 5MB. When the query coordinator process combines the sorted subindexes, some of the extents may be trimmed, and the resulting index may be smaller than the requested 60MB.

When you add or enable a UNIQUE key or PRIMARY KEY constraint on a table, you cannot automatically create the required index in parallel. Instead, manually create an index on the desired columns using the CREATE INDEX statement and an appropriate PARALLEL clause and then add or enable the constraint. Oracle then uses the existing index when enabling or adding the constraint.

Multiple constraints on the same table can be enabled concurrently and in parallel if all the constraints are already in the enabled novalidate state. In the following example, the ALTER TABLE ... ENABLE CONSTRAINT statement performs the table scan that checks the constraint in parallel:

```
CREATE TABLE a (a1 NUMBER CONSTRAINT ach CHECK (a1 > 0) ENABLE NOVALIDATE)
PARALLEL;
INSERT INTO a values (1);
  COMMIT;
   ALTER TABLE a ENABLE CONSTRAINT ach;
```

See Also: For more information on how extents are allocated when using the parallel execution feature, see *Oracle8i Concepts*. Also refer to the *Oracle8i SQL Reference* for the complete syntax of the CREATE INDEX statement.

Parallel DML Tips

This section provides an overview of parallel DML functionality.

- INSERT
- Direct-Load INSERT
- Parallelizing INSERT, UPDATE, and DELETE

See Also: *Oracle8i Concepts* for a detailed discussion of parallel DML and DOP. For a discussion of parallel DML affinity, please see *Oracle8i Parallel Server Concepts and Administration*.

INSERT

Oracle INSERT functionality can be summarized as follows:

Table 27–3 Summary of INSERT Features

Insert Type	Parallel	Serial	NOLOGGING
Conventional	No	Yes	No
Direct Load Insert (Append)	Yes: requires: ▪ ALTER SESSION ENABLE PARALLEL DML ▪ Table PARALLEL attribute or PARALLEL hint ▪ APPEND hint (optional)	Yes: requires: ▪ APPEND hint	Yes: requires: ▪ NOLOGGING attribute set for table or partition

If parallel DML is enabled and there is a PARALLEL hint or PARALLEL attribute set for the table in the data dictionary, then inserts are parallel and appended, unless a restriction applies. If either the PARALLEL hint or PARALLEL attribute is missing, then the insert is performed serially.

Direct-Load INSERT

Append mode is the default during a parallel insert: data is always inserted into a new block which is allocated to the table. Therefore the APPEND hint is optional. You should use append mode to increase the speed of insert operations—but not when space utilization needs to be optimized. You can use NOAPPEND to override append mode.

The APPEND hint applies to both serial and parallel insert: even serial inserts are faster if you use this hint. APPEND, however, does require more space and locking overhead.

You can use NOLOGGING with APPEND to make the process even faster. NOLOGGING means that no redo log is generated for the operation. NOLOGGING is never the default; use it when you wish to optimize performance. It should not normally be used when recovery is needed for the table or partition. If recovery is needed, be sure to take a backup immediately after the operation. Use the ALTER TABLE [NO]LOGGING statement to set the appropriate value.

See Also: *Oracle8i Concepts.*

Parallelizing INSERT, UPDATE, and DELETE

When the table or partition has the PARALLEL attribute in the data dictionary, that attribute setting is used to determine parallelism of INSERT, UPDATE, and DELETE statements as well as queries. An explicit PARALLEL hint for a table in a statement overrides the effect of the PARALLEL attribute in the data dictionary.

You can use the NOPARALLEL hint to override a PARALLEL attribute for the table in the data dictionary. In general, hints take precedence over attributes.

DML operations are considered for parallelization only if the session is in a PARALLEL DML enabled mode. (Use ALTER SESSION ENABLE PARALLEL DML to enter this mode.) The mode does not affect parallelization of queries or of the query portions of a DML statement.

See Also: *Oracle8i Concepts* for more information on parallel INSERT, UPDATE and DELETE.

Parallelizing INSERT ... SELECT In the INSERT... SELECT statement you can specify a PARALLEL hint after the INSERT keyword, in addition to the hint after the SELECT keyword. The PARALLEL hint after the INSERT keyword applies to the insert operation only, and the PARALLEL hint after the SELECT keyword applies to the select operation only. Thus parallelism of the INSERT and SELECT operations are independent of each other. If one operation cannot be performed in parallel, it has no effect on whether the other operation can be performed in parallel.

The ability to parallelize INSERT causes a change in existing behavior, if the user has explicitly enabled the session for parallel DML, and if the table in question has a

PARALLEL attribute set in the data dictionary entry. In that case existing INSERT ... SELECT statements that have the select operation parallelized may also have their insert operation parallelized.

If you query multiple tables, you can specify multiple SELECT PARALLEL hints and multiple PARALLEL attributes.

Example

Add the new employees who were hired after the acquisition of ACME.

```
INSERT /*+ PARALLEL(EMP) */ INTO EMP
SELECT /*+ PARALLEL(ACME_EMP) */ *
FROM ACME_EMP;
```

The APPEND keyword is not required in this example, because it is implied by the PARALLEL hint.

Parallelizing UPDATE and DELETE The PARALLEL hint (placed immediately after the UPDATE or DELETE keyword) applies not only to the underlying scan operation, but also to the update/delete operation. Alternatively, you can specify update/delete parallelism in the PARALLEL clause specified in the definition of the table to be modified.

If you have explicitly enabled PDML (Parallel Data Manipulation Language) for the session or transaction, UPDATE/DELETE statements that have their query operation parallelized may also have their UPDATE/DELETE operation parallelized. Any subqueries or updatable views in the statement may have their own separate parallel hints or clauses, but these parallel directives do not affect the decision to parallelize the update or delete. If these operations cannot be performed in parallel, it has no effect on whether the UPDATE or DELETE portion can be performed in parallel.

You can only use parallel UPDATE and DELETE on partitioned tables.

Example 1

Give a 10% salary raise to all clerks in Dallas.

```
UPDATE /*+ PARALLEL(EMP) */ EMP
SET SAL=SAL * 1.1
  WHERE JOB='CLERK' AND
  DEPTNO IN
  (SELECT DEPTNO FROM DEPT WHERE LOCATION='DALLAS');
```

The PARALLEL hint is applied to the update operation as well as to the scan.

Example 2

Fire all employees in the accounting department, whose work will be outsourced.

```
DELETE /*+ PARALLEL(EMP) */ FROM EMP
WHERE DEPTNO IN
(SELECT DEPTNO FROM DEPT WHERE DNAME='ACCOUNTING');
```

Again, the parallelism is applied to the scan as well as update operation on table EMP.

Refreshing Tables in Parallel

Parallel DML combined with the updatable join views facility provides an efficient solution for refreshing the tables of a data warehouse system. To refresh tables is to update them with the differential data generated from the OLTP production system.

In the following example, assume that you want to refresh a table named CUSTOMER(c_key, c_name, c_addr). The differential data contains either new rows or rows that have been updated since the last refresh of the data warehouse. In this example, the updated data is shipped from the production system to the data warehouse system by means of ASCII files. These files must be loaded into a temporary table, named DIFF_CUSTOMER, before starting the refresh process. You can use SQL Loader with both the parallel and direct options to efficiently perform this task.

Once DIFF_CUSTOMER is loaded, the refresh process can be started. It is performed in two phases:

- Updating the table
- Inserting the new rows in parallel

Updating the Table

A straightforward SQL implementation of the update uses subqueries:

```
UPDATE CUSTOMER
SET(C_NAME, C_ADDR) =
(SELECT C_NAME, C_ADDR
  FROM DIFF_CUSTOMER
    WHERE DIFF_CUSTOMER.C_KEY = CUSTOMER.C_KEY)
    WHERE C_KEY IN(SELECT C_KEY FROM DIFF_CUSTOMER);
```

Unfortunately, the two subqueries in the preceding statement affect the performance.

An alternative is to rewrite this query using updatable join views. To do this you must first add a primary key constraint to the DIFF_CUSTOMER table to ensure that the modified columns map to a key-preserved table:

```
CREATE UNIQUE INDEX DIFF_PKEY_IND ON DIFF_CUSTOMER(C_KEY)
 PARALLEL NOLOGGING;
  ALTER TABLE DIFF_CUSTOMER ADD PRIMARY KEY (C_KEY);
```

Update the CUSTOMER table with the following SQL statement:

```
UPDATE /*+ PARALLEL(CUST_JOINVIEW) */
(SELECT /*+ PARALLEL(CUSTOMER) PARALLEL(DIFF_CUSTOMER) */
CUSTOMER.C_NAME as C_NAME
CUSTOMER.C_ADDR as C_ADDR,
DIFF_CUSTOMER.C_NAME as C_NEWNAME,
DIFF_CUSTOMER.C_ADDR as C_NEWADDR
  WHERE CUSTOMER.C_KEY = DIFF_CUSTOMER.C_KEY) CUST_JOINVIEW
  SET C_NAME = C_NEWNAME, C_ADDR = C_NEWADDR;
```

The base scans feeding the join view CUST_JOINVIEW are done in parallel. You can then parallelize the update to further improve performance but only if the CUSTOMER table is partitioned.

> **See Also:** "Rewriting SQL Statements" on page 27-18. Also see the *Oracle8i Application Developer's Guide - Fundamentals* for information about key-preserved tables.

Inserting the New Rows into the Table in Parallel

The last phase of the refresh process consists in inserting the new rows from the DIFF_CUSTOMER to the CUSTOMER table. Unlike the update case, you cannot avoid having a subquery in the insert statement:

```
INSERT /*+PARALLEL(CUSTOMER)*/ INTO CUSTOMER
SELECT * FROM DIFF_CUSTOMER
  WHERE DIFF_CUSTOMER.C_KEY NOT IN (SELECT /*+ HASH_AJ */ KEY FROM CUSTOMER);
```

But here, the HASH_AJ hint transforms the subquery into an anti-hash join. (The hint is not required if the parameter ALWAYS_ANTI_JOIN is set to hash in the initialization file). Doing so allows you to use parallel insert to execute the preceding statement very efficiently. Parallel insert is applicable even if the table is not partitioned.

Using Hints with Cost Based Optimization

Cost-based optimization is a highly sophisticated approach to finding the best execution plan for SQL statements. Oracle automatically uses cost-based optimization with parallel execution.

Note: You must use ANALYZE to gather current statistics for cost-based optimization. In particular, tables used in parallel should always be analyzed. Always keep your statistics current by running ANALYZE after DDL and DML operations.

Use discretion in employing hints. If used, hints should come as a final step in tuning, and only when they demonstrate a necessary and significant performance advantage. In such cases, begin with the execution plan recommended by cost-based optimization, and go on to test the effect of hints only after you have quantified your performance expectations. Remember that hints are powerful; if you use them and the underlying data changes you may need to change the hints. Otherwise, the effectiveness of your execution plans may deteriorate.

Always use cost-based optimization unless you have an existing application that has been hand-tuned for rule-based optimization. If you must use rule-based optimization, rewriting a SQL statement can greatly improve application performance.

Note: If any table in a query has a DOP greater than one (including the default DOP), Oracle uses the cost-based optimizer for that query—even if OPTIMIZER_MODE = RULE, or if there is a RULE hint in the query itself.

See Also: "OPTIMIZER_PERCENT_PARALLEL" on page 26-23. This parameter controls parallel awareness.

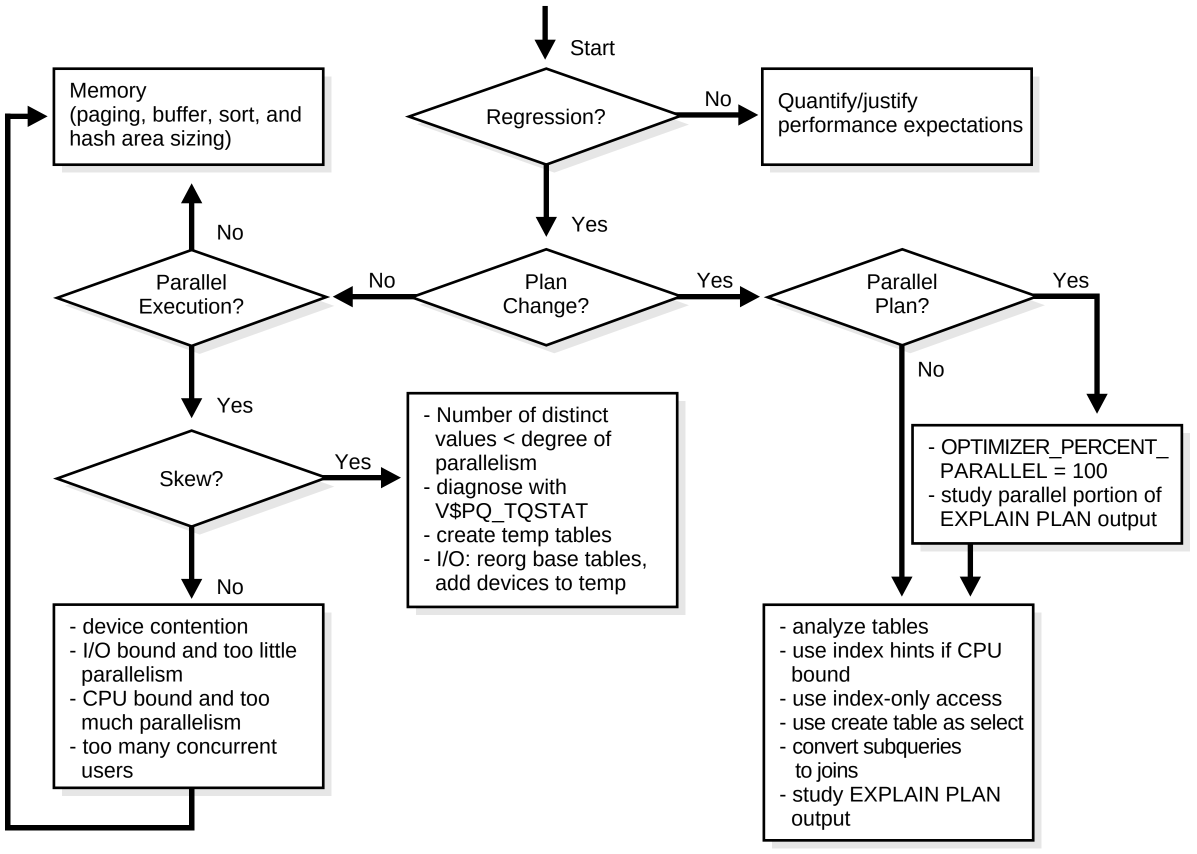
Diagnosing Problems

Use the decision tree in Figure 27–6 to diagnose parallel performance problems. The questions in the decision points of Figure 27–6 are discussed in more detail after the figure.

Some key issues in diagnosing parallel execution performance problems are the following:

- Quantify your performance expectations to determine whether there is a problem.
- Determine whether a problem pertains to optimization, such as inefficient plans that may require re-analyzing tables or adding hints, or whether the problem pertains to execution, such as simple operations like scanning, loading, grouping, or indexing running much slower than published guidelines.
- Determine whether the problem occurs when running in parallel, such as load imbalance or resource bottlenecks, or whether the problem is also present for serial operations.

Figure 27–6 Parallel Execution Performance Checklist


Start
Regression?
No
Quantify/justify performance expectations
Yes
Memory (paging, buffer, sort, and hash area sizing)
No
Parallel Execution?
No
Plan Change?
Yes
Parallel Plan?
Yes
No
Yes
Skew?
Yes
- Number of distinct values < degree of parallelism
- diagnose with V$PQ_TQSTAT
- create temp tables
- I/O: reorg base tables, add devices to temp
- OPTIMIZER_PERCENT_ PARALLEL = 100
- study parallel portion of EXPLAIN PLAN output
No
- device contention
- I/O bound and too little parallelism
- CPU bound and too much parallelism
- too many concurrent users
- analyze tables
- use index hints if CPU bound
- use index-only access
- use create table as select
- convert subqueries to joins
- study EXPLAIN PLAN output

Is There Regression?

Does parallel execution’s actual performance deviate from what you expected? If performance is as you expected, could there be an underlying performance problem? Perhaps you have a desired outcome in mind to which you are comparing the current outcome. Perhaps you have justifiable performance expectations that the system does not achieve. You might have achieved this level of performance or particular execution plan in the past, but now, with a similar environment and operation, your system is not meeting this goal.

If performance is not as you expected, can you quantify the deviation? For data warehousing operations, the execution plan is key. For critical data warehousing operations, save the EXPLAIN PLAN results. Then, as you analyze the data, reanalyze, upgrade Oracle, and load new data, over time you can compare new execution plans with old plans. Take this approach either proactively or reactively.

Alternatively, you may find that plan performance improves if you use hints. You may want to understand why hints were necessary, and determine how to get the optimizer to generate the desired plan without the hints. Try increasing the statistical sample size: better statistics may give you a better plan. If you had to use a PARALLEL hint, determine whether you had OPTIMIZER_PERCENT_PARALLEL set to 100%.

See Also: For more information on the EXPLAIN PLAN statement, refer to Chapter 13, "Using EXPLAIN PLAN". For information on preserving plans throughout changes to your system using Plan Stability and outlines, please refer to Chapter 7, "Optimizer Modes, Plan Stability, and Hints".

Is There a Plan Change?

If there has been a change in the execution plan, determine whether the plan is (or should be) parallel or serial.

Is There a Parallel Plan?

If the execution plan is or should be parallel:

- Try increasing OPTIMIZER_PERCENT_PARALLEL to 100 if you want a parallel plan, but the optimizer has not given you one.
- Study the EXPLAIN PLAN output. Did you analyze *all* the tables? Perhaps you need to use hints in a few cases. Verify that the hint provides better performance.

See Also: Parallel EXPLAIN PLAN tags are defined in Table 13–2.

Is There a Serial Plan?

If the execution plan is or should be serial, consider the following strategies:

- Use an index. Sometimes adding an index can greatly improve performance. Consider adding an extra column to the index: perhaps your operation could obtain all its data from the index, and not require a table scan. Perhaps you need to use hints in a few cases. Verify that the hint gives better results.

- If you do not analyze often, and you can spare the time, it is a good practice to compute statistics. This is particularly important if you are performing many joins and it will result in better plans. Alternatively, you can estimate statistics.

Note: Using different sample sizes can cause the plan to change. Generally, the higher the sample size, the better the plan.

- Use histograms for non-uniform distributions.
- Check initialization parameters to be sure the values are reasonable.
- Replace bind variables with literals.
- Determine whether execution is I/O or CPU bound. Then check the optimizer cost model.
- Convert subqueries to joins.
- Use CREATE TABLE ... AS SELECT to break a complex operation into smaller pieces. With a large query referencing five or six tables, it may be difficult to determine which part of the query is taking the most time. You can isolate bottlenecks in the query by breaking it into steps and analyzing each step.

See Also: *Oracle8i Concepts* regarding CREATE TABLE ... AS SELECT.

Is There Parallel Execution?

If the cause of regression cannot be traced to problems in the plan, then the problem must be an execution issue. For data warehousing operations, both serial and parallel, consider how your plan uses memory. Check the paging rate and make sure the system is using memory as effectively as possible. Check buffer, sort, and hash area sizing. After you run a query or DML operation, look at the V$SESSTAT, V$PX_SESSTAT, and V$PQ_SYSSTAT views to see the number of server processes used and other information for the session and system.

See Also: "Monitoring Parallel Execution Performance with Dynamic Performance Views" on page 26-78.

Is The Workload Evenly Distributed?

If you are using parallel execution, is there unevenness in workload distribution? For example, if there are 10 CPUs and a single user, you can see whether the workload is evenly distributed across CPUs. This may vary over time, with periods that are more or less I/O intensive, but in general each CPU should have roughly the same amount of activity.

The statistics in V$PQ_TQSTAT show rows produced and consumed per parallel execution server. This is a good indication of skew and does not require single user operation.

Operating system statistics show you the per-processor CPU utilization and per-disk I/O activity. Concurrently running tasks make it harder to see what is going on, however. It can be useful to run in single-user mode and check operating system monitors that show system level CPU and I/O activity.

When workload distribution is unbalanced, a common culprit is the presence of skew in the data. For a hash join, this may be the case if the number of distinct values is less than the degree of parallelism. When joining two tables on a column with only 4 distinct values, you will not get scaling on more than 4. If you have 10 CPUs, 4 of them will be saturated but 6 will be idle. To avoid this problem, change the query: use temporary tables to change the join order such that all operations have more values in the join column than the number of CPUs.

If I/O problems occur you may need to reorganize your data, spreading it over more devices. If parallel execution problems occur, check to be sure you have followed the recommendation to spread data over at least as many devices as CPUs.

If there is no skew in workload distribution, check for the following conditions:

- Is there device contention? Are there enough disk controllers to provide adequate I/O bandwidth?
- Is the system I/O bound, with too little parallelism? If so, consider increasing parallelism up to the number of devices.
- Is the system CPU bound, with too much parallelism? Check the operating system CPU monitor to see whether a lot of time is being spent in system calls. The resource may be overcommitted, and too much parallelism may cause processes to compete with themselves.
- Are there more concurrent users than the system can support?

Part VI

Materialized Views

Part Six discusses materialized views. The chapters in Part Six are:

28 Data Warehousing with Materialized Views

This chapter contains:

Overview of Data Warehousing with Materialized Views

An enterprise *data warehouse* contains historical detailed data about the organization. Typically, data flows from one or more online transaction processing (OLTP) databases into the data warehouse on a monthly, weekly, or daily basis. The data is usually processed in a *staging file* before being added to the data warehouse. Data warehouses typically range in size from tens of gigabytes to a few terabytes, usually with the vast majority of the data stored in a few very large fact tables.

A *data mart* contains a subset of corporate data that is of value to a specific business unit, department, or set of users. Typically, a data mart is derived from an enterprise data warehouse.

One of the techniques employed in data warehouses to improve performance is the creation of summaries, or aggregates. They are a special kind of aggregate view which improves query execution times by precalculating expensive joins and aggregation operations prior to execution, and storing the results in a table in the database. For example, a table may be created which would contain the sum of sales by region and by product.

Today, organizations using summaries spend a significant amount of time manually creating summaries, identifying which ones to create, indexing the summaries, updating them, and advising their users on which ones to use. The introduction of summary management in the Oracle server changes the workload of the DBA dramatically and means the end-user no longer has to be aware of which summaries have been defined. The DBA creates one or more materialized views, which are the equivalent of a summary. The end-user queries the tables and views in the database and the query rewrite mechanism in the Oracle server automatically rewrites the SQL query to use the summary tables. This results in a significant improvement in response time for returning results from the query and eliminates the need for the end-user or database application to be aware of the summaries that exist within the data warehouse.

Although summaries are usually accessed indirectly via the query rewrite mechanism, an end-user or database application can construct queries which directly access the summaries. However, serious consideration should be given to whether users should be allowed to do this because, once the summaries are directly referenced in queries, the DBA will not be free to drop and create summaries without affecting applications.

The summaries or aggregates that are referred to in this book and in literature on data warehousing are created in Oracle using a schema object called a materialized view. Materialized views can be used to perform a number of roles, such as improving query performance or providing replicated data, as described below.

Materialized Views for Data Warehouses

In data warehouses, materialized views can be used to precompute and store aggregated data such as sum of sales. Materialized views in these environments are typically referred to as summaries since they store summarized data. They can also be used to precompute joins with or without aggregations. So a materialized view is used to eliminate overhead associated with expensive joins or aggregations for a large or important class of queries.

Materialized Views for Distributed Computing

In distributed environments, materialized views are used to replicate data at distributed sites and synchronize updates done at several sites with conflict resolution methods. The materialized views as replicas provide local access to data which otherwise would have to be accessed from remote sites.

Materialized Views for Mobile Computing

Materialized views are used to download a subset of data from central servers to mobile clients, with periodic refreshes from the central servers and propagation of updates by clients back to the central servers.

This chapter is focused on the use of materialized views in data warehouses. Refer to *Oracle8i Replication* and *Oracle8i Distributed Database Systems* for details on distributed and mobile computing.

Components of Summary Management

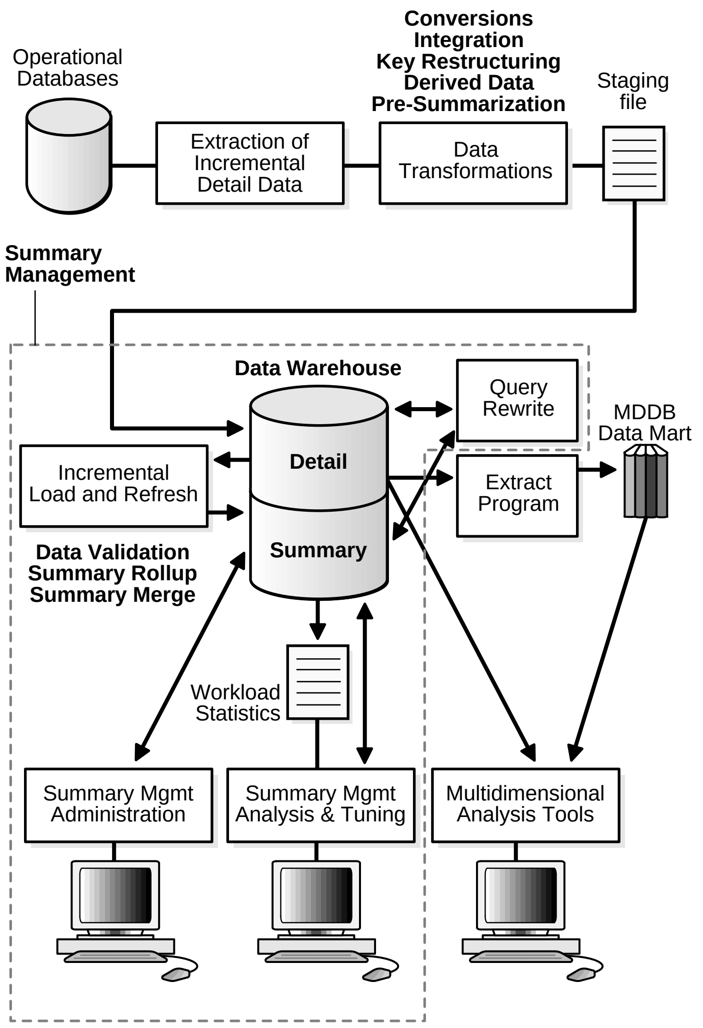
Summary management consists of:

- Mechanisms to define summaries and dimensions
- A refresh mechanism to ensure that all summaries contain the latest data
- A query rewrite capability to transparently rewrite a query to use a summary
- An advisor utility to recommend which summaries to create, retain, and drop

Many large decision support system (DSS) databases have schemas that do not closely resemble a conventional data warehouse schema, but still require joins and aggregates. The use of summary management features imposes no schema restrictions, and may enable some existing DSS database applications to achieve large gains in performance without requiring a redesign of the database or application. This functionality is thus available to all database users.

Figure 28–1 illustrates where summary management is used in the warehousing cycle. It is available once the data has been transformed and loaded into the data warehouse. Therefore, referring to Figure 28–1, after the data has been transformed, staged, and loaded into the detail data in the warehouse, then the summary management process can be invoked. This means that summaries can be created, queries can be rewritten, and the advisor can be used to plan summary usage and creation.

Figure 28–1 Overview of Summary Management



Understanding the summary management process during the earliest stages of data warehouse design can yield large dividends later on in the form of higher performance, lower summary administration costs, and reduced storage requirements.

The summary management process begins with the creation of dimensions and hierarchies that describe the business relationships and common access patterns in the database. An analysis of the dimensions, combined with an understanding of the typical work load, can then be used to create materialized views. Materialized views improve query execution performance by pre-calculating expensive join or aggregation operations prior to execution time. Query rewrite then automatically recognizes when an existing materialized view can and should be used to satisfy a request, and can transparently rewrite a request to use a materialized view, thus improving performance.

Other considerations when building a warehouse include:

- Horizontally partitioning the fact tables by a time attribute.

  This improves scalabililty, simplifies system administration, and makes it possible to define local indexes that can be efficiently rebuilt.

- SQL*Loader can be directed to load a single partition of a table.

  In this case, only the corresponding local index partitions are rebuilt.

- Global indexes must be fully rebuilt after a direct load, which can be very costly when loading a relatively small number of rows into a large table.

  For this reason, it is strongly recommended that all fact table indexes should be defined as local indexes. For example, this can be accomplished by having a bitmap index on each key column (bitmap indexes are always local), and a single multi-key index that includes all the key columns, with the partitioning attribute as the leading column of the index.

Terminology

The following clarifies some basic data warehousing terms:

- *Dimension tables* describe the business entities of an enterprise, which usually represent hierarchical, categorical information such as time, departments, locations, and products. Dimension tables are sometimes called *lookup or reference tables*.

  Dimension tables usually change slowly over time and are not modified on a periodic schedule. They are typically not large, but they affect the performance of long-running decision support queries that consist of joins of fact tables with dimension tables, followed by aggregation to specific levels of the dimension hierarchies.

- *Fact tables* describe the business transactions of an enterprise. Fact tables are sometimes called *detail tables*.

  The vast majority of data in a data warehouse is stored in a few very large fact tables. They are updated periodically with data from one or more operational online transaction processing (OLTP) databases.

  Fact tables include *measures* such as sales, units, and inventory.

  – A simple measure is a numeric or character column of one table such as FACT.SALES.

  – A computed measure is an expression involving only simple measures of one table, for example, FACT.REVENUES - FACT.EXPENSES.

  – A multi-table measure is a computed measure defined on multiple tables, for example, FACT_A.REVENUES - FACT_B.EXPENSES.

  Fact tables also contain one or more *keys* that organize the business transactions by the relevant business entities such as time, product, and market. In most cases, the fact keys are non-null, form a unique compound key of the fact table, and join with one and only one row of a *dimension table*.

- *A materialized view* is a pre-computed table comprising aggregated and/or joined data from fact and possibly dimension tables. Builders of data warehouses will know a materialized view as a *summary* or *aggregation*.

Materialized Views

The most common situations where you would find materialized views useful are in data warehousing applications and distributed systems. In warehousing applications, large amounts of data are processed and similar queries are frequently repeated. If these queries are pre-computed and the results stored in the data warehouse as a materialized view, using materialized views significantly improves performance by providing fast lookups into the set of results.

A materialized view definition can include any number of aggregates, as well as any number of joins. In several ways, a materialized view behaves like an index:

- The purpose of a materialized view is to increase request execution performance.
- The existence of a materialized view is transparent to SQL applications, so a DBA can create or drop materialized views at any time without affecting the validity of SQL applications.

- A materialized view consumes storage space.
- The contents of the materialized view must be maintained when the underlying detail tables are modified.

This chapter shows how materialized views are used in a data warehousing environment. However, the materialized view that is a key component of summary management can also be used in a distributed environment to manage replicated data. For further information, see *Oracle8i Replication*.

Schema Design Guidelines for Materialized Views

Before starting to define and use the various components of summary management, it is recommended that you review your schema design to, wherever possible, abide by these guidelines:

Guideline 1: Your dimensions should either be denormalized (each dimension contained in one table) or the joins between tables in a normalized or partially normalized dimension should guarantee that each child-side row joins with one and only one parent-side row. The benefits of maintaining this condition are described in "Creating a Dimension" on page 30-3.

If desired, this condition can be enforced by adding `FOREIGN KEY` and `NOT NULL` constraints on the child-side join key(s) and `PRIMARY KEY` constraints on the parent-side join key(s). If your materialized view contains only a single detail table, or if it performs no aggregation, a preferred alternative would be to use outer joins in place of inner joins. In this case, the Oracle optimizer can guarantee the integrity of the result without enforced referential integrity constraints.

Guideline 2: If dimensions are denormalized or partially denormalized, hierarchical integrity must be maintained between the key columns of the dimension table. Each child key value must *uniquely identify* its parent key value, even if the dimension table is denormalized. Hierarchical integrity in a denormalized dimension can be verified by calling the `VALIDATE_DIMENSION` procedure of the `DBMS_OLAP` package.

Guideline 3: Fact tables and dimension tables should similarly guarantee that each fact table row joins with one and only one dimension table row. This condition must be declared, and optionally enforced, by adding FOREIGN KEY and NOT NULL constraints on the fact key column(s) and PRIMARY KEY constraints on the dimension key column(s), or by using outer joins as described in Guideline 1. In a data warehouse, constraints are typically enabled with the NOVALIDATE and RELY options to avoid constraint enforcement performance overhead.

Guideline 4: Incremental loads of your detail data should be done using the SQL*Loader direct-path option, or any bulk loader utility that uses Oracle's direct path interface (including INSERT AS SELECT with the APPEND or PARALLEL hints). If the materialized view contains more than one table and performs aggregation, or if materialized view logs are not defined, then performing any other type of DML to your data will necessitate a complete refresh.

Guideline 5: Horizontally partition your tables by a monotonically increasing time column if possible (preferably of type DATE). For each table, create a bitmap index for each key column, and create one local index that includes all the key columns. Stripe each horizontal partition across several storage devices for maximum performance.

Guideline 6: After each load and *before* refreshing your materialized view, use the VALIDATE_DIMENSION procedure of the DBMS_OLAP package to incrementally verify dimensional integrity.

Guideline 7: Horizontally partition and index the materialized view like the fact tables. Include a local concatenated index on all the materialized view keys.

While guidelines 1, 2, and 3 are each important during schema design, guidelines 1 and 2 are even more important than guideline 3. If your schema design does not follow guidelines 1 and 2, it does not then matter whether it follows guideline 3. Guidelines 1, 2, and 3 affect both query rewrite performance and materialized view refresh performance. Guideline 4 affects materialized view refresh performance only. If your schema design does not follow guideline 4, then incremental refresh of your materialized views will be either impossible or much less efficient.

If you are concerned with the time required to enable constraints and whether any constraints may be violated, use the ENABLE NOVALIDATE clause to turn on constraint checking without validating any of the existing constraints. The risk with this approach is that incorrect query results could occur if any constraints are broken. Therefore, this is a decision for the designer to determine how clean the data is and whether the risk of potential wrong results is too great.

Materialized view management can perform many useful functions, including query rewrite and materialized view refresh, even if your data warehouse design does not follow these guidelines; however, you will realize significantly greater query execution performance and materialized view refresh performance benefits, and you will require fewer materialized views if your schema design complies with these guidelines.

Oracle Tools for Data Warehousing

The availability of powerful tools to help automate the analysis and administration of the materialized views is an important factor in controlling data warehouse costs. The following Oracle tools are available to help you create and manage a data warehouse:

Data Mart Designer or *Oracle Designer* can be used to design the warehouse schema. Data is then extracted, transformed, and transferred *(ETT)* from the operational systems into the data warehouse or data mart. *Data Mart Builder* can be used to specify the ETT process, populate the target data mart, and automatically schedule loads and index rebuilds.

Discoverer can be used to query the database and queries executed via Discoverer will be rewritten when appropriate. The Discoverer summary wizard can be used to recommend which materialized views to create because Discoverer retains its own workload statistics with respect to query usage.

The data mart may be analyzed natively with Discoverer or it can be optionally exported to the *Express* multidimensional database server through the *Relational Access Manager (RAM)*. Analysis of the data in Express supports reach-through to detail data stored in the Oracle8*i* server through RAM, and provides relational access to tools like *Oracle Sales Analyzer (OSA)* and *Oracle Express Objects (OEO)*.

Getting Started

The following chapters describe how to create materialized views and dimensions. Although materialized views can be created at any time, so that they can used by the other features in summary management such as warehouse refresh and query rewrite, some parameters must be set. These can be defined either within the initialization parameter file or using the ALTER SYSTEM or ALTER SESSION commands. The required parameters are identified by subject area.

- **Warehouse Refresh**

  JOB_QUEUE_PROCESSES

    The number of background processes. This parameter determines how many materialized views can be refreshed concurrently.

  JOB_QUEUE_INTERVAL

    In seconds, the interval between which the job queue scheduler checks to see if a new job has been submitted to the job queue.

  UTL_FILE_DIR

    The directory where the refresh log is written. If unspecified, no refresh log will be created.

- **Query Rewrite**

  OPTIMIZER_MODE="ALL_ROWS", "FIRST_ROWS", or "CHOOSE"

    With tables analyzed, ensures that the cost-based optimizer is used, which is a requirement to get Query Rewrite.

  QUERY_REWRITE_ENABLED = True

    Turns on query rewrite.

  QUERY_REWRITE_INTEGRITY = enforced or trusted or stale_tolerated

    Optional. Advises how fresh a materialized view must be to be eligible for query rewrite. See *Oracle8i Reference* for further information about the values for QUERY_REWRITE_INTEGRITY.

  COMPATIBLE

    Must be 8.1 or higher.

- **Advisor Workload**

Recommended Parameters:

  ORACLE_TRACE_COLLECTION_NAME = oraclsm

    Trace collection file name.

  ORACLE_TRACE_COLLECTION_PATH = ?/otrace/admin/cdf

    Location where the collection file is stored.

ORACLE_TRACE_COLLECTION_SIZE = 0

Initial size of the collection file.

Required Parameters and their Settings:

ORACLE_TRACE_ENABLE=true

Turns on Oracle Trace collection.

ORACLE_TRACE_FACILITY_NAME = oraclesm

Trace facility to collect data.

ORACLE_TRACE_FACILITY_PATH = ?/otrace/admin/cdf

Location of the Trace facility definition files.

- **Recommended Parameters for Parallelism**

PARALLEL_MAX_SERVERS

Should be set high enough to take care of parallelism.

SORT_AREA_SIZE

Should be less than HASH_AREA_SIZE.

OPTIMIZER_MODE

Should equal CHOOSE (cost based optimizer).

OPTIMIZER_PERCENT_PARALLEL

Should equal 100.

Once these parameters have been set to the appropriate values, you will be ready to move on to using the summary management features.

29 Materialized Views

The materialized views introduced in Oracle8*i* are generic objects that are used to summarize, precompute, replicate, and distribute data. They are suitable in various computing environments such as data warehousing, decision support, and distributed, or mobile computing.

Several new functional areas have been developed to offer comprehensive and robust support for the management and use of materialized views in different computing environments. The new functionality includes transparent query rewrite, object dependency management, staleness tracking of materialized data, new refresh methods such as transactionally consistent refresh on commit, and highly efficient incremental fast refresh using direct path and DML logs.

This chapter contains:

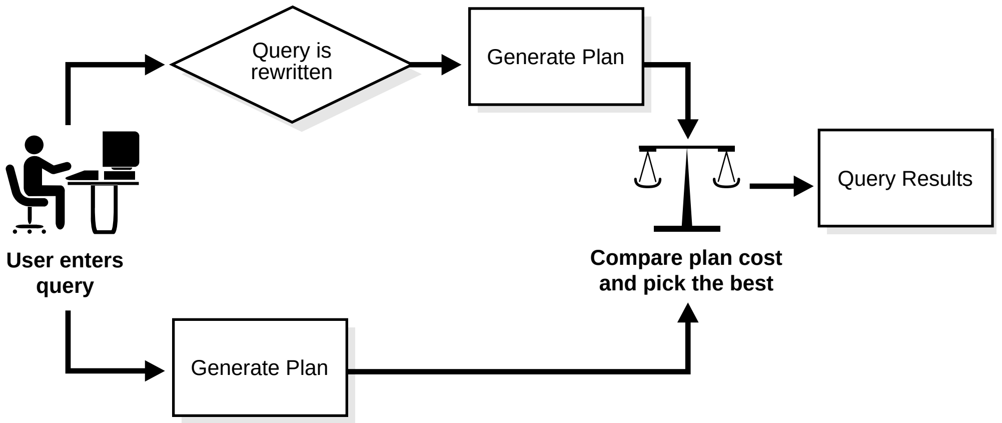
The Need for Materialized Views

Materialized views are used in warehouses to increase the speed of queries on very large databases. Queries to large databases often involve joins between tables or aggregations such as SUM, or both. These operations are very expensive in terms of time and processing power. The type of materialized view that is created determines how it can be refreshed and used by query rewrite.

Materialized views can be used in a number of ways and almost identical syntax can be used to perform a number of roles. For example, a materialized view can be used to replicate data, which was formerly achieved by using the CREATE SNAPSHOT command. Now CREATE MATERIALIZED VIEW is a synonym for CREATE SNAPSHOT.

Materialized views improve query performance by precalculating expensive join and aggregation operations on the database prior to execution time and storing these results in the database. The query optimizer can make use of materialized views by automatically recognizing when an existing materialized view can and should be used to satisfy a request. It then transparently rewrites the request to use the materialized view. Queries are then directed to the materialized view and not to the underlying detail tables or views. Rewriting queries to use materialized views rather than detail relations results in a significant performance gain.

Figure 29–1 Transparent Query Rewrite



Thus, when using query rewrite, you want to create materialized views that satisfy the largest number of queries. For example, if you identify twenty queries that are commonly applied to the detail or fact tables, then you might be able to satisfy them with five or six well-written materialized views. A materialized view definition can include any number of aggregations (SUM, COUNT(x), COUNT(*), COUNT(DISTINCT x), AVG, VARIANCE, STDDEV, MIN, and MAX) and/or include any number of joins. In case you are unsure of which materialized views to create, Oracle provides a set of advisory functions in the DBMS_OLAP package to help in designing and evaluating materialized views for query rewrites.

If a materialized view is to be used by query rewrite, it must be stored in the same database as its fact or detail tables. A materialized view can be partitioned, and you can define a materialized view on a partitioned table and one or more indexes on the materialized view.

Materialized views are similar to indexes in several ways: they consume storage space, they must be refreshed when the data in their master tables changes, and, when used for query rewrites, they improve the performance of SQL execution and their existence is transparent to SQL applications and users. Unlike indexes, materialized views can be accessed directly using a SELECT statement and, depending on the types of refresh that are required, they can also be accessed directly in an INSERT, UPDATE, or DELETE statement.

Note: Materialized views can also be used by Oracle Replication. The techniques shown in this chapter illustrate how to use materialized views in data warehouses.

Creating a Materialized View

To create a materialized view, use the CREATE MATERIALIZED VIEW command. The following command creates the materialized view *store_sales_mv*.

```
CREATE MATERIALIZED VIEW store_sales_mv
  PCTFREE 0 TABLESPACE mviews
  STORAGE (INITIAL 16k NEXT 16k PCTINCREASE 0)
  PARALLEL
  BUILD DEFERRED
  REFRESH COMPLETE
  ENABLE QUERY REWRITE
  AS
  SELECT
   s.store_name,
     SUM(dollar_sales) AS sum_dollar_sales
      FROM store s, fact f
      WHERE f.store_key = s.store_key
```

```
GROUP BY s.store_name;
```

See Also: For a complete description of `CREATE MATERIALIZED VIEW`, see the *Oracle8i SQL Reference*.

It is not uncommon in a data warehouse to have already created summary or aggregation tables, and the DBA may not wish to repeat this work by building a new materialized view. In this instance, the table that already exists in the database can be registered as a *prebuilt* materialized view. This technique is described in "Registration of an Existing Materialized View" on page 29-16.

Once you have selected the materialized views you want to create, follow the steps below for each materialized view.

1. Do the physical design of the materialized view (existing user-defined materialized views do not require this step). The materialized view should be horizontally partitioned by a time attribute (if possible) and should match the partitioning of the largest or most frequently updated detail or fact table (if possible). Refresh performance generally benefits from a large number of horizontal partitions because it can take advantage of the parallel capabilities in Oracle.
2. Use the `CREATE MATERIALIZED VIEW` statement to create and, optionally, populate the materialized view. If a user-defined materialized view already exists, then use the `PREBUILT` option in the `CREATE MATERIALIZED VIEW` statement. Otherwise, use the `BUILD IMMEDIATE` option to populate the materialized view immediately, or the `BUILD DEFERRED` option to populate the materialized view at a more convenient time (the materialized view is disabled for use by query rewrite until the first `REFRESH`, after which it will be automatically enabled, provided the ENABLE QUERY REWRITE clause has been specified).

   See Also: See *Oracle8i SQL Reference* for descriptions of the SQL statements CREATE MATERIALIZED VIEW, ALTER MATERIALIZED VIEW, and DROP MATERIALIZED VIEW.

Naming

The name given to a materialized view must conform to standard Oracle naming conventions. However, if the materialized view is based on a user-defined prebuilt table, then the name of the materialized view must exactly match that table name.

If you already have a naming convention for tables and indexes, you may consider extending this naming scheme to the materialized views so that they are easily identifiable. For example, instead of naming the materialized view *sum_of_sales*, it could be called *sum_of_sales_mv* to denote that this is a materialized view and not a table or view, for instance.

Storage Characteristics

Unless the materialized view is based on a user-defined prebuilt table, it requires and occupies storage space inside the database. Therefore, the storage needs for the materialized view should be specified in terms of the tablespace where it is to reside and the size of the extents.

If you do not know how much space the materialized view will require, then the DBMS_OLAP.ESTIMATE_SIZE package, which is described in "Summary Advisor" on page 32-14, can provide an estimate on the bytes required to store this materialized view. This information can then assist the design team in determining into which tablespace the materialized view should reside.

> **See Also:** For a complete description of the `STORAGE` semantics, see the *Oracle8i SQL Reference*.

Build Methods

Two build methods are available for creating the materialized view, as shown in the table below. If you select BUILD IMMEDIATE, the materialized view definition is added to the schema objects in the data dictionary, and then the fact or detail tables are scanned as per the SELECT expression and the results are stored in the materialized view. Depending on the size of the tables to be scanned, this build process can take a considerable amount of time.

An alternative approach is to use the BUILD DEFERRED clause, which creates the materialized view without data, thereby enabling it to be populated at a later date using the DBMS_MVIEW.REFRESH package described in "Warehouse Refresh" on page 32-3.

Build Method	Description
BUILD DEFERRED	Create the materialized view definition but do not populate it with data.
BUILD IMMEDIATE	Create the materialized view and then populate it with data.

Used for Query Rewrite

When a materialized view is defined, it will not automatically be used by the query rewrite facility. Therefore, the clause ENABLE QUERY REWRITE must be specified if the materialized view is to be considered available for rewriting queries.

If this clause is omitted or specified as DISABLE QUERY REWRITE when the materialized view is initially created, the materialized view can subsequently be enabled for query rewrite with the `ALTER MATERIALIZED VIEW` statement.

Query Rewrite Restrictions

Query rewrite is not possible with all materialized views. If query rewrite is not occurring when it was expected, check to see if your materialized view does not satisfy one of the following conditions.

Materialized View Restrictions

1. There cannot be non-repeatable expressions anywhere in the defining query (ROWNUM, SYSDATE, non-repeatable PL/SQL functions, and so on).
2. There cannot be references to RAW or LONG RAW datatypes or object REFs.
3. The query must be a single-block query, that is, it cannot contain set functions (UNION, MINUS, and so on). However, a materialized view can have multiple query blocks (e.g., inline views in the FROM clause and subselects in the WHERE or HAVING clauses).
4. If the materialized view was registered as PREBUILT, the precision of columns must agree with the precision of the corresponding SELECT expressions unless overridden by WITH REDUCED PRECISION.

Query Rewrite Restrictions

1. Detail tables must be local—only local detail tables or views can be accessed in the query or used in the definition of the materialized view.
2. None of the detail tables can be owned by SYS, and the materialized view cannot be owned by SYS.

Non-SQL Text Rewrite Restrictions

1. The FROM list cannot contain multiple occurrences of the same table or view.

2. SELECT and GROUP BY lists, if present, must be the same in the query and the materialized view and must contain straight columns, that is, no expressions are allowed in the columns.
3. Aggregate operators must occur only as the outermost part of the expression, that is, aggregates such as AVG(AVG(x)) or AVG(x)+AVG(x) are not allowed.
4. The WHERE clause must contain only inner or outer equijoins, which can be connected by ANDs. That is, no ORs and no selections on single tables are allowed in the WHERE clause.
5. HAVING or CONNECT BY clauses are not allowed.

Refresh Options

If you are going to refresh your materialized views from the detail or fact tables, then you must add a `REFRESH` clause to the `CREATE MATERIALIZED VIEW` statement. When defining the refresh clause, two elements need to be specified: what type of refresh should occur and how to execute the refresh.

The two refresh execution modes are: ON COMMIT and ON DEMAND. The method you select will determine the type of materialized view that can be defined.

Refresh Mode	Description
`ON COMMIT`	Refresh occurs automatically on the next `COMMIT` performed at the master table. Can be used with materialized views on single table aggregates and materialized views containing joins only.
`ON DEMAND`	Refresh occurs when a user manually executes one of the available refresh procedures contained in the `DBMS_MVIEW` package (`REFRESH`, `REFRESH_ALL_MVIEWS`, `REFRESH_DEPENDENT`).

If the materialized view is being refreshed using the ON COMMIT method, then, following refresh operations, the alert log and trace file should be consulted to check that no errors have occurred.

If a materialized view fails during refresh at COMMIT time, the user has to explicitly invoke the refresh procedure using the DBMS_MVIEW package after addressing the errors specified in the trace files. Until this is done, the view will no longer be refreshed automatically at commit time.

Selecting the ON DEMAND execution mode means that you can take advantage of the materialized view *warehouse refresh* facility, which provides a quick and efficient mechanism for refreshing your materialized views, either in their entirety or only with the additions to the detail data.

You can specify how you want your materialized views to be refreshed from the detail tables by selecting one of four options: `FORCE`, `COMPLETE`, `FAST`, and `NEVER`.

Refresh Option	Description
`COMPLETE`	Refreshes by recalculating the materialized view's defining query when ATOMIC REFRESH=TRUE and COMPLETE is the same as FORCE if ATOMIC REFRESH=FALSE.
`FAST`	Refreshes by incrementally adding the new data that has been inserted into the tables using direct path or from the materialized view log.
`FORCE`	First determines if fast refresh is possible and applies it if it is; otherwise, it applies `COMPLETE` refresh.
`NEVER`	Suppresses refresh of the materialized view.

Whether the fast refresh option is available will depend upon the type of materialized view that has been created. The table below summarizes under what conditions fast refresh is possible for the different types of materialized views. Creation of the materialized view will fail and an error will be reported if these conditions are not met.

Table 29–1 Requirements for Fast Refresh of Materialized Views

	When the Materialized View has:		
	Only Joins	Joins and Aggregates	Aggregate on a Single Table
Detail tables only	X	X	X
Single table only	-	-	X
Table Appears only once in the FROM list	X	X	X
No non-repeating expressions like SYSDATE and ROWNUM	X	X	X
No references to RAW or LONG RAW	X	X	X
No GROUP BY	X	-	-
Rowids of all the detail tables must appear in the SELECT list of the query	X	-	-

Table 29–1 Requirements for Fast Refresh of Materialized Views

Expressions are allowed in the GROUP BY and SELECT clauses provided they are the same	-	X	X
Aggregates allowed but cannot be nested	-	X	X
AVG with COUNT	-	X	X
SUM with COUNT	-	-	X
VARIANCE with COUNT and SUM	-	X	X
STDDEV with COUNT and SUM	-	X	X
WHERE clause contains join predicates which can be ANDed bit not ORed.	X	X	-
No WHERE clause	-	-	X
No HAVING or CONNECT BY	X	X	X
No subqueries, inline views, or set functions like UNION or MINUS	X	X	X
COUNT(*) must be present	-	-	X
No MIN and MAX allowed	-	-	X
If outer joins, then unique constraints must exist on the join columns of the inner join table	X	-	-
Materialized View logs must exist and contain all columns referenced in the materialized view and have been created with the LOG NEW VALUES clause	-	-	X
Materialized view logs must exist with rowids of all the detail tables	X	-	-
Non-aggregate expression in SELECT and GROUP BY must be straight columns	-	-	X
DML to detail table	X	-	X
Direct path data load	X	X	X
ON COMMIT	X	-	X
ON DEMAND	X	X	X

Defining the Data for the Materialized View

The SELECT clause in the materialized view defines the data that it is to contain and there are only a few restrictions on what may be specified. Any number of tables may be joined together, however, they cannot be remote tables if you wish to take advantage of query rewrite or the warehouse refresh facility. It is not only tables that may be joined or referenced in the SELECT clause, because views, inline views, subqueries and materialized views are all permissible.

Materialized Views with Joins and Aggregates

In data warehouses, materialized views would normally contain one of the aggregates shown in the table below. To get warehouse incremental refresh, the `SELECT` list must contain all of the `GROUP BY` columns (if present), and may contain one or more aggregate functions. The aggregate function must be one of: `SUM`, `COUNT(x)`, `COUNT(*)`,`COUNT(DISTINCT x)`, `AVG`, `VARIANCE`, `STDDEV`, `MIN`, and `MAX`, and the expression to be aggregated can be any SQL value expression.

Here are some examples of the type of materialized view which can be created.

Create Materialized View: Example 1

```
CREATE MATERIALIZED VIEW store_sales_mv
  PCTFREE 0 TABLESPACE mviews
  STORAGE (initial 16k next 16k pctincrease 0)
  BUILD DEFERRED
  REFRESH COMPLETE ON DEMAND
  ENABLE QUERY REWRITE
  AS
  SELECT
   s.store_name,
     SUM(dollar_sales) AS sum_dollar_sales
      FROM store s, fact f
      WHERE f.store_key = s.store_key
      GROUP BY s.store_name;
```

The statement above creates a materialized view *store_sales_mv* that computes the sum of *sales* by *store*. It is derived by joining the tables *store* and *fact* on the column *store_key*. The materialized view does not initially contain any data because the build method is DEFERRED. When it is refreshed, a complete refresh is performed and, once populated, this materialized view can be used by query rewrite.

Create Materialized View: Example 2

```
CREATE MATERIALIZED VIEW store_avgcnt_mv
```

```
  PCTFREE 0 TABLESPACE mviews
  STORAGE (INITIAL 16k NEXT 16k PCTINCREASE 0)
  BUILD IMMEDIATE
  REFRESH COMPLETE
  ENABLE QUERY REWRITE
  AS
  SELECT store_name,
     AVG(unit_sales) AS avgcnt_unit_sales,
     COUNT(DISTINCT(f.time_key)) AS count_days
      FROM store s, fact f, time t
      WHERE s.store_key = f.store_key AND
            f.time_key = t.time_key
      GROUP BY store_name, t.time_key;
```

The statement above creates a materialized view *store_avgcnt_mv* that computes the average number of units sold by a *store* on a given date. It is derived by joining the tables *store, time,* and *fact* on the columns *store_key* and *time_key*. The materialized view is populated with data immediately because the build method is IMMEDIATE and it is available for use by query rewrite. Note that the ON DEMAND clause has been omitted from this materialized view definition because it is optional; because it is the default, the materialized view will not be refreshed until a manual request is made.

Create Materialized View: Example 3

```
CREATE MATERIALIZED VIEW store_stdcnt_mv
  PCTFREE 0 TABLESPACE mviews
  STORAGE (INITIAL 16k NEXT 16k PCTINCREASE 0)
  BUILD IMMEDIATE
  REFRESH FAST
  ENABLE QUERY REWRITE
  AS
  SELECT store_name, t.time_key,
     STDDEV(unit_sales) AS stdcnt_unit_sales
     AVG(unit_sales) AS avgcnt_unit_sales
     COUNT(unit_sales) AS count_days
     SUM(unit_sales) AS sum_unit_sales
   FROM store s, fact f, time t
      WHERE s.store_key = f.store_key AND
            f.time_key = t.time_key
      GROUP BY store_name, t.time_key;
```

The statement above creates a materialized view *store_stdcnt_mv* that computes the standard deviation for the number of units sold by a *store* on a given date. It is derived by joining the tables *store*, *time* and *fact* on the column store_key and time_key. The materialized view is populated with data immediately because the build method is immediate and it is available for use by query rewrite. In this example, the refresh method is FAST, which is allowed because the COUNT and SUM aggregates have been included to support fast refresh of the STDDEV aggregate.

Create Materialized View: Example 4

```
CREATE MATERIALIZED VIEW store_var_mv
 PCTFREE 0  TABLESPACE mviews
   STORAGE (INITIAL 16k NEXT 16k PCTINCREASE 0)
  PARALLEL
  BUILD DEFERRED
  REFRESH FORCE
    AS
    SELECT s.store_key, store_name,
           VARIANCE(unit_sales) AS var_unit_sales
     FROM fact f, store s, time t
     WHERE s.store_key = f.store_key  AND
           f.time_key = t.time_key
        GROUP BY s.store_key, t.time_key, store_name;
```

The statement above creates a materialized view *store_stdcnt_mv* that computes the variance for the number of units sold by a *store* on a given *date*. It is derived by joining the tables *store*, *time*, and *fact* on the columns *store_key* and *time_key*. The materialized view is not populated with data immediately and the materialized view is not available for use by query rewrite because the ENABLE QUERY REWRITE clause has not been specified. The refresh method is FORCE, which means that the most suitable refresh method will be selected.

Single Table Aggregate Materialized Views

A materialized view which contains one or more aggregates (SUM, AVG, VARIANCE, STDDEV, COUNT) and a GROUP BY may be based on a single table. The aggregate function can involve an expression on the columns such as SUM(a*b). If this materialized view is to be incrementally refreshed, then a materialized view log must be created on the detail table which includes the INCLUDING NEW VALUES option, and contains all columns referenced in the materialized view query definition.

In this release, it is assumed that the materialized view and all the base tables the materialized view is dependent upon must belong to the same schema.

```
CREATE MATERIALIZED VIEW log on fact
  with rowid (store_key, time_key, dollar_sales, unit_sales)
  including new values;

CREATE MATERIALIZED VIEW sum_sales
  PARALLEL
  BUILD IMMEDIATE
  REFRESH FAST ON COMMIT
  AS
  SELECT f.store_key, f.time_key,
         COUNT(*) AS count_grp,
SUM(f.dollar_sales) AS sum_dollar_sales,
        COUNT(f.dollar_sales) AS count_dollar_sales,
SUM(f.unit_sales) AS sum_unit_sales,
        COUNT(f.unit_sales) AS count_unit_sales
  FROM fact f
  GROUP BY f.store_key, f.time_key;
```

In this example, a materialized view has been created which contains aggregates on a single table. Because the materialized view log has been created, the materialized view is fast refreshable. Whenever DML is applied against the fact table, when the commit is issued, the changes will be reflected in the materialized view.

Table 29–2 illustrates the aggregate requirements for a single table aggregate materialized view.

Table 29–2 Single Table Aggregate Requirements

If aggregate X is present, aggregate Y is required and aggregate Z is optional		
X	**Y**	**Z**
`COUNT(expr)`	-	-
`SUM(expr)`	`COUNT(expr)`	-
`AVG(expr)`	`COUNT(expr)`	`SUM(expr)`
`STDDEV(expr)`	`COUNT(expr)`	`SUM(expr * expr)`
`VAR(expr)`	`COUNT(expr)`	`SUM(expr * expr)`

Note that COUNT(*) must always be present.

Materialized Views Containing Only Joins

Materialized views may contain only joins and no aggregates, such as in the next example where a materialized view is created which joins the *fact* to the *store* table.

The advantage of creating this type of materialized view is that expensive joins have been precalculated.

If you specify REFRESH FAST, Oracle performs further verification of the query definition to ensure that fast refresh can always be performed if **any** of the detail tables change. These additional checks include:

1. A materialized view log must be present for each detail table.
2. The rowids of all the detail tables must appear in the SELECT list of the materialized view query definition.
3. If there are outer joins, unique constraints must be on the join columns of the inner table.

   For example, if you are joining the fact and a dimension table and the join is an outer join with the fact table being the outer table, there must exist unique constraints on the join columns of the dimension table.

If some of the above restrictions are not met, then the materialized view must be created as REFRESH FORCE. If one of the tables did not meet all of the criteria, but the other tables did, the materialized view would still be incrementally refreshable, but only for the other tables for which all the criteria are met.

In this release, it is assumed that the materialized view and all the base tables the materialized view is dependent upon must belong to the same schema.

In a data warehouse star schema, if space is at a premium, you can include the rowid of the fact table only as this is the table that will be most frequently updated, and the user can specify the FORCE option when the materialized view is created.

A materialized view log should contain the rowid of the master table. It is not necessary to add other columns.

Incremental refresh for a materialized view containing only joins is possible after any type of DML to the base tables (direct load or conventional INSERT, UPDATE, or DELETE).

A materialized view containing only joins can be defined to be refreshed ON COMMIT or ON DEMAND. If it is ON COMMIT, the refresh is performed at commit time of the transaction that does DML on the detail table.

After a refresh on-commit, you are urged to check the alert log and trace files to see if any error occurred during the refresh.

To speed up refresh, it is recommended that the user create indices on the columns of the materialized view that stores the rowids of the fact table.

```
CREATE MATERIALIZED VIEW LOG ON fact
  WITH ROWID;

CREATE MATERIALIZED VIEW LOG ON time
  WITH ROWID;

CREATE MATERIALIZED VIEW LOG ON store
  WITH ROWID;

CREATE MATERIALIZED VIEW detail_fact_mv
       PARALLEL
       BUILD IMMEDIATE
       REFRESH FAST
       AS
       SELECT
    f.rowid "fact_rid", t.rowid "time_rid", s.rowid "store_rid",
      s.store_key, s.store_name, f.dollar_sales,
      f.unit_sales, f.time_key
       FROM fact f, time t, store s
       WHERE f.store_key = s.store_key(+) AND
       f.time_key = t.time_key(+);
```

In the example shown above, in order to perform a REFRESH FAST, unique constraints should exist on *s.store_key* and *t.time_key*. It is also recommended that indexes be created on columns *fact_rid*, *time_rid*, and *store_rid*, as illustrated below, which will improve the performance of refresh.

```
CREATE INDEX mv_ix_factrid  ON
  detail_fact_mv(fact_rid);
```

Alternatively, if the example shown above did not include the columns *time_rid* and *store_rid*, and if the refresh method was REFRESH FORCE, then this materialized view would be fast refreshable if the *fact* table changed but not if the tables *time* or *store* changed.

```
CREATE MATERIALIZED VIEW detail_fact_mv
       PARALLEL
       BUILD IMMEDIATE
       REFRESH FORCE
       AS
       SELECT
    f.rowid "fact_rid",
      s.store_key, s.store_name, f.dollar_sales,
      f.unit_sales, f.time_key
       FROM fact f, time t, store s
```

```
 WHERE f.store_key = s.store_key(+) AND
f.time_key = t.time_key(+);
```

Registration of an Existing Materialized View

Some data warehouses have implemented materialized views in ordinary user tables. Although this solution provides the performance benefits of materialized views, it does not provide query rewrite to all SQL applications, does not enable materialized views defined in one application to be transparently accessed in another application, and does not generally support fast parallel or fast incremental materialized view refresh.

Because of these problems, and because existing materialized views may be extremely large and expensive to rebuild, you should register your existing materialized view tables with the Oracle server whenever possible. You can register a user-defined materialized view with the `CREATE MATERIALIZED VIEW` ... `ON PREBUILT TABLE` statement. Once registered, the materialized view can be used for query rewrites or maintained by one of the refresh methods, or both.

In some cases, user-defined materialized views are refreshed on a schedule that is longer than the update cycle; for example, a monthly materialized view may be updated only at the end of each month, and the materialized view values always refer to *complete time periods*. Reports written directly against these materialized views implicitly select only data that is not in the current (incomplete) time period. If a user-defined materialized view already contains a time dimension:

- It should be registered and then incrementally refreshed each update cycle.
- A view should be created that selects the complete time period of interest.

  For example, if a materialized view was formerly refreshed monthly at the end of each month, then the view would contain the selection `WHERE` *time.month* < `CURRENT_MONTH`().
- The reports should be modified to refer to the view instead of referring directly to the user-defined materialized view.

If the user-defined materialized view does not contain a time dimension, then:

- A new materialized view should be created that does include the time dimension (if possible).
- The view should aggregate over the time column in the new materialized view.

The table must reflect the materialization of the defining query at the time you register it as a materialized view, and each column in the defining query must correspond to a column in the table that has a matching datatype. However, you can specify WITH REDUCED PRECISION to allow the precision of columns in the defining query to be different from that of the table columns.

The table and the materialized view must have the same name, but the table retains its identity as a table and can contain columns that are not referenced in the defining query of the materialized view *(unmanaged columns)*. If rows are inserted during a refresh operation, each unmanaged column of the row is set to its default value, therefore the unmanaged columns cannot have NOT NULL constraints unless they also have default values.

Unmanaged columns are not supported by single table aggregate materialized views or materialized views containing joins only.

Materialized views based on prebuilt tables are eligible for selection by query rewrite provided the parameter QUERY_REWRITE_INTEGRITY is set to at least the level of TRUSTED. See Chapter 31, "Query Rewrite" for details about integrity levels.

When you drop a materialized view that was created on a prebuilt table, the table still exists—only the materialized view is dropped.

When a prebuilt table is registered as a materialized view, the parameter QUERY_REWRITE_INTEGRITY must be set to at least STALE_TOLERATED because, when it is created, the materialized view is marked as stale, therefore, only stale integrity modes can be used.

```
CREATE TABLE sum_sales_tab
  PCTFREE 0  TABLESPACE mviews
   STORAGE (INITIAL 16k NEXT 16k PCTINCREASE 0)
    AS
    SELECT f.store_key
       SUM(dollar_sales) AS dollar_sales,
       SUM(unit_sales) AS unit_sales,
       SUM(dollar_cost) AS dollar_cost
         FROM fact f GROUP BY f.store_key;

CREATE MATERIALIZED VIEW sum_sales_tab
ON PREBUILT TABLE WITHOUT REDUCED PRECISION
ENABLE QUERY REWRITE
AS
SELECT f.store_key,
  SUM(dollar_sales) AS dollar_sales,
  SUM(unit_sales) AS unit_sales,
  SUM(dollar_cost) AS dollar_cost
  FROM fact f GROUP BY f.store_key;
```

This example illustrates the two steps required to register a user-defined table. First, the table must be created, then the materialized view is defined using exactly the same name as the table. This materialized view *sum_sales_tab* is eligible for use in query rewrite.

Partitioning a Materialized View

Due to the large volume of data held in a data warehouse, partitioning is an extremely useful option that can be used by the database designer.

Horizontally partitioning the fact tables by a time attribute improves scalability, simplifies system administration, and makes it possible to define local indexes that can be efficiently rebuilt. SQL*Loader can be directed to load a single partition of a table. In this case, only the corresponding local index partitions are rebuilt. Global indexes must be fully rebuilt after a direct load, which can be very costly when loading a relatively small number of rows into a large table. For this reason, it is strongly recommended that all fact table indexes should be defined as local indexes. For example, this can be accomplished by having a bitmap index on each key column (bitmap indexes are always local) and a single multikey index that includes all the key columns, with the partitioning attribute as the leading column of the multikey index.

Partitioning a materialized view also has benefits as far as refresh is concerned, since the refresh procedure can use parallel DML to maintain the materialized view. To realize these benefits, the materialized view has to be defined as PARALLEL and parallel DML must be enabled in the session.

When the data warehouse or data mart contains a time dimension, it is often desirable to archive the oldest information, and then reuse the storage for new information. If the fact tables or materialized views include a time dimension and are horizontally partitioned by the time attribute, then management of rolling materialized views can be reduced to a few fast partition maintenance operations provided that the unit of data that is rolled out equals, or is at least aligned with, the horizontal partitions.

If you plan to have rolling materialized views in your warehouse, then you should determine how frequently you plan to perform partition maintenance operations, and you should plan to horizontally partition fact tables and materialized views to reduce the amount of system administration overhead required when old data is aged out.

With the introduction of new partitioning options in Oracle8*i*, you are not restricted to using range partitions. For example, a composite partition using both a time value and, say, a store_key value could result in an ideal partition solution for your data.

For further details about partitioning, see *Oracle8i Concepts*.

An ideal case for using partitions is when a materialized view contains a subset of the data, which is obtained by defining an expression of the form WHERE time_key < '1-OCT-1998' in the SELECT expression for the materialized view. If a WHERE clause of this type is included, then query rewrite will be restricted to the *exact match* case, which severely restricts when the materialized view is used. To overcome this problem, use a partitioned materialized view with no WHERE clause and then query rewrite will be able to use the materialized view and it will only search the appropriate partition, thus improving query performance.

There are two approaches to partitioning a materialized view:

- Partitioning the Materialized View
- Partitioning the Prebuilt Table

Partitioning the Materialized View

Partitioning a materialized view involves defining the materialized view with the standard Oracle partitioning clauses as illustrated in the example below. This example creates a materialized view called *part_sales_mv* which uses three partitions, is fast refreshed, and is eligible for query rewrite.

```
CREATE MATERIALIZED VIEW part_sales_mv
  PARALLEL
  PARTITION by RANGE (time_key)
  (
    PARTITION time_key
      VALUES LESS THAN (TO_DATE('31-12-1997', 'DD-MM-YYYY'))
      PCTFREE 0 PCTUSED
      STORAGE (INITIAL 64k NEXT 16k PCTINCREASE 0)
      TABLESPACE sf1,
   PARTITION month2
     VALUES LESS THAN (TO_DATE('31-01-1998', 'DD-MM-YYYY'))
     PCTFREE 0 PCTUSED
     STORAGE INITIAL 64k NEXT 16k PCTINCREASE 0)
      TABLESPACE sf2,
 PARTITION month3
     VALUES LESS THAN (TO_DATE('31-01-1998', 'DD-MM-YYYY'))
     PCTFREE 0 PCTUSED
     STORAGE (INITIAL 64k NEXT 16k PCTINCREASE 0)
      TABLESPACE sf3)
BUILD DEFERRED
REFRESH FAST
ENABLE QUERY REWRITE
AS
SELECT f.store_key, f.time_key,
  SUM(f.dollar_sales) AS sum_dol_sales,
       SUM(f.unit_sales) AS sum_unit_sales
         FROM fact f GROUP BY f.time_key, f.store_key;
```

Partitioning a Prebuilt Table

Alternatively, a materialized view can be registered to a partitioned prebuilt table as illustrated below.

```
CREATE TABLE part_fact_tab(
       time_key, store_key, sum_dollar_sales,
         sum_unit_sale)
  PARALLEL
  PARTITION by RANGE (time_key)
  (
    PARTITION month1
      VALUES LESS THAN (TO_DATE('31-12-1997', 'DD-MM-YYYY'))
      PCTFREE 0 PCTUSED 99
      STORAGE (INITITAL 64k NEXT 16k PCTINCREASE 0)
      TABLESPACE sf1,
   PARTITIION month2
     VALUES LESS THAN (TO_DATE('31-01-1998', 'DD-MM-YYYY'))
     PCTFREE 0 PCTUSED 99
     STORAGE (INITIAL 64k NEXT 16k PCTINCREASE 0)
      TABLESPACE sf2,
 PARTITION month3
     VALUES LESS THAN (TO_DATE('31-01-1998', DD-MM-YYYY'))
     PCTFREE 0 PCTUSED 99
     STORAGE INITIAL 64k NEXT 16k PCTINCREASE 0)
     TABLESPACE sf3)
AS
SELECT f.time_key, f.store_key,
  SUM(f.dollar_sales) AS sum_dollar_sales,
  SUM(f.unit_sales)   AS sum_unit_sales
         FROM fact f GROUP BY f.time_key, f.store_key;

CREATE MATERIALIZED VIEW part_fact_tab
```

```
ON PREBUILT TABLE
ENABLE QUERY REWRITE
AS
SELECT f.time_key,  f.store_key,
  SUM(f.dollar_sales) AS sum_dollar_sales,
  SUM(f.unit_sales)   AS sum_unit_sales
        FROM fact f  GROUP BY  f.time_key , f.store_key;
```

In this example, the table *part_fact_tab* has been partitioned over three months and then the materialized view was registered to use the prebuilt table. This materialized view is eligible for query rewrite because the ENABLE QUERY REWRITE clause has been included.

Indexing Selection for Materialized Views

The two main operations on a materialized view are query execution and fast incremental refresh, and each operation has different performance requirements. Fast incremental refresh needs to perform an exact match on the materialized view keys, and performs best when there is a concatenated index that includes all of the materialized view keys. Query execution, on the other hand, may need to access any subset of the materialized view key columns, and may need to join and aggregate over a subset of those columns; consequently, query execution usually performs best if there is a single-column bitmap index defined on each materialized view key column.

One option for indexing the materialized view is to define a unique, local index that contains all of the materialized view keys, and a single-column bitmap index on each materialized view key, if storage space and refresh time permit.

In the case of materialized views containing joins only using the fast refresh option, it is highly recommended that indexes be created on the columns which contain the rowids to improve the performance of the refresh operation.

Invalidating a Materialized View

Dependencies related to materialized views are automatically maintained to ensure correct operation. At DDL time, a materialized view depends on the detail tables referenced in its definition.

A shared cursor depends on all objects referenced in the cursor. If a cursor is rewritten, the cursor depends on the materialized view selected by query rewrite and the dimensions of the tables of the cursor if they are being used by query rewrite. Any operation that would invalidate these dimensions or the materialized view would invalidate the cursor.

Therefore, any DDL operation, such as a DROP or ALTER, on any dependency in the materialized view will cause it to become invalid.

The state of a materialized view can be checked by querying the table USER_MVIEW_ANALYSIS or ALL_MVIEW_ANALYSIS. The column UNUSABLE takes a value of Y or N and advises whether the materialized view may be used. The column KNOWN_STALE also takes a value of Y or N and advises whether a materialized view is known to be stale and finally column INVALID will be set to Y if the materialized view is invalid and N if it is not.

A materialized view is automatically revalidated whenever it is referenced. However, if a column has been dropped in a table referenced by a materialized view or the owner of the materialized view didn't have one of the query rewrite privileges and that has now been granted to them, the command

```
ALTER MATERIALIZED VIEW  mview_name ENABLE QUERY REWRITE
```

should be used to revalidate the materialized view and, if there are any problems, an error will be returned.

Security Issues

To create a materialized view, the privilege CREATE MATERIALIZED VIEW is required, and to create a materialized view that references a table in another schema, the privilege CREATE ANY MATERIALIZED VIEW is needed.

If the materialized view is to be used by query rewrite, then the privilege QUERY REWRITE should be granted, or if the materialized view references tables not in your schema, then GLOBAL QUERY REWRITE must be granted.

If you continue to get a privilege error while trying to create a materialized view and you believe that all the required privileges have been granted, then the problem is most likely due to a privilege not being granted explicitly and it has been inherited from a role instead. The owner of the materialized view must have explicitly been granted SELECT access to the referenced tables.

Guidelines for using Materialized Views in a Data Warehouse

Determining what materialized views would be most beneficial, in terms of performance gains, is aided by the analysis tools of the `DBMS_OLAP` package. Specifically, you can call the `DBMS_OLAP.RECOMMEND_MV` procedure to see a list of materialized views that Oracle recommends based on the statistics and the usage of the target database. Note that this package currently only recommends materialized views having aggregates on multiple tables.

If you are going to write your own materialized views without the aid of Oracle analysis tools, then use these guidelines to achieve maximum performance:

1. Instead of defining multiple materialized views on the same tables with the same `GROUP BY` columns but with different measures, define a single materialized view including all of the different measures.
2. If your materialized view includes the aggregated measure `AVG`(x), also include `COUNT`(x) to support incremental refresh. Similarly, if `VARIANCE`(x) or `STDDEV`(x) is present, then always include `COUNT`(x) and `SUM`(x) to support incremental refresh.

Altering a Materialized View

There are only three amendments that can be made to a materialized view:

- change its refresh method
- recompile
- enable / disable its use for query rewrite

All other changes are achieved by dropping and then recreating the materialized view.

The COMPILE option of the `ALTER MATERIALIZED VIEW` statement can be used when the materialized view has been invalidated as described in "Invalidating a Materialized View" on page 29-21. This compile process is quick, which means that the materialized view can be used by query rewrite.

For further information about ALTER MATERIALIZED VIEW, see *Oracle8i SQL Reference.*

Dropping a Materialized View

Use the `DROP MATERIALIZED VIEW` command to drop a materialized view. For example:

```
DROP MATERIALIZED VIEW sales_sum_mv;
```

This command drops the materialized view `sales_sum_mv`. If the materialized view was prebuilt on a table, then the table is not dropped but it can no longer be maintained with the refresh mechanism.

30 Dimensions

This chapter contains:

Dimensions in a Data Warehouse

Dimensions do not have to be defined, but spending time creating them can yield significant benefits because they help query rewrite perform more complex types of rewrite. They are mandatory if you use the advisor to recommend which materialized views to create, drop, or retain.

A business process is an operational process within an organization about which data can be collected. As an example, each store of a video chain might gather and store data regarding sales and rentals of video tapes at the check-out counter. The video chain management can build a data warehouse to analyze the sales of its products across all stores over time and help answer questions such as:

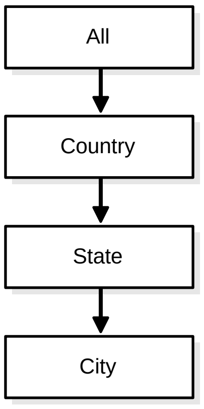
- What is the effect of promoting one product on the sale of a related product that is not promoted?
- What are the product sales before and after the promotion?

The data in the video chain's data warehouse system has two important components: *dimensions* and *facts*. The dimensions are products, locations (stores), promotions, and time. One approach for identifying your dimensions is to review your reference tables, such as a product table which contains everything about a product, or a store table containing all information about a store. The facts are sales (units sold or rented) and profits. A data warehouse contains facts about the sales of each product at each store on a daily basis.

Dimension values are usually organized into hierarchies. Going up a level in the hierarchy is called *rolling up* the data and going down a level in the hierarchy is called *drilling down* the data. In the video chain example:

- Within the time dimension, months roll up to quarters, quarters roll up to years, and years roll up to all years.
- Within the product dimension, products roll up to categories, categories roll up to departments, and departments roll up to all departments.
- Within the location dimension, stores roll up to cities, cities roll up to states, states roll up to regions, regions roll up to countries, and countries roll up to all countries, as shown in Figure 30–1.

Figure 30–1 Geography Dimension



Data analysis typically starts at higher levels in the dimensional hierarchy and gradually drills down if the situation warrants such analysis.

You can visualize the dimensions of a business process as an n-dimensional data cube. In the video chain example, the business dimensions product, location, and time can be represented along the three axes of the cube. Each unit along the product axis represents a different product, each unit along the location axis represents a store, and each unit along the time axis represents a month. At the intersection of these values is a cell that contains factual information, such as units sold and profits made. Higher-level analysis consists of selecting and aggregating the factual information within a subcube, such as rentals of comedy videos in California stores during the second quarter of 1998.

Therefore, the first step towards creating a dimension is to identify the dimensions within your data warehouse and then draw the hierarchies as shown in Figure 30–1. For example, *city* is a child of *state* (because you can aggregate city-level data up to state), and *state*. Using this approach, you should find it easier to translate this into an actual dimension.

In the case of normalized or partially normalized dimensions (a dimension that is stored in more than one table), identify how these tables are joined. Note whether the joins between the dimension tables can guarantee that each child-side row joins with one and only one parent-side row. In the case of denormalized dimensions, determine whether the child-side columns uniquely determine the parent-side (or attribute) columns. These constraints can be enabled with the `NOVALIDATE` and `RELY` options if the relationships represented by the constraints are guaranteed by other means. Note that if the joins between fact and dimension tables do not support this relationship, you still gain significant performance advantages from defining the dimension with the `CREATE DIMENSION` statement. Another alternative, subject to certain restrictions, is to use outer joins in the materialized view definition (that is, in the `CREATE MATERIALIZED VIEW` statement).

You must not create dimensions in any schema that does not satisfy these relationships, incorrect results can be returned from queries otherwise.

Creating a Dimension

Before you can create a dimension, tables must exist in the database which contain this dimension data. For example, if you create a dimension called LOCATION, one or more tables must exist which contains the city, state, and country information. In a data warehouse, these dimension tables already exist. It is therefore a simple task to identify which ones will be used.

You create a dimension with the `CREATE DIMENSION` statement. Within the `CREATE DIMENSION` statement, use the `LEVEL...IS` clause to identify the names of the dimension levels.

The location dimension contains a single hierarchy, with arrows drawn from the *child level* to the *parent level*. At the top of this dimension graph is the special level `ALL`, that represents aggregation over all rows. Each arrow in this graph indicates that for any child there is one and only one parent. For example, each city must be contained in exactly one state and each state must be contained in exactly one country. States that belong to more than one country, or that belong to no country, violate hierarchical integrity. Hierarchical integrity is necessary for the correct operation of management functions for materialized views that include aggregates.

Therefore, using the entities illustrated in Figure 30–1 on page 30-2 as an example, you can declare a dimension LOCATION which contains levels CITY, STATE, and COUNTRY:

```
CREATE DIMENSION location_dim
    LEVEL city      IS location.city
    LEVEL state     IS location.state
    LEVEL country   IS location.country
```

Using your drawing of the dimension, translate each level in the diagram to a LEVEL clause in the CREATE DIMENSION statement. You therefore define 3 levels: city, state, and country. Then each level in the dimension must correspond to one or more columns in a table in the database. Thus, level *city* is identified by the column *city* in the table called *location* and level *country* is identified by a column called *country* in the same table.

In this example, the database tables are denormalized and all the columns exist in the same table. However, this is not a prerequisite for creating dimensions. "Using Normalized Dimension Tables" on page 30-7 shows how to create a dimension that has a normalized schema design using the JOIN KEY clause.

The next step is to declare the relationship between the levels with the HIERARCHY statement and give that hierarchy a name. A hierarchical relationship is a *functional dependency* from one level of a hierarchy to the next level in the hierarchy. Using the *level names* defined previously, the CHILD OF relationship denotes that each child's level value is associated with one and only one parent level value. Again, using the entities in Figure 30–1 on page 30-2, the following statements declare a hierarchy LOC_ROLLUP and define the relationship between CITY, STATE, and COUNTRY.

```
HIERARCHY loc_rollup    (
        city     CHILD OF
        state    CHILD OF
        country     )
```

In addition to the 1:n hierarchical relationships, dimensions also include 1:1 attribute relationships between the hierarchy levels and their dependent dimension attributes. For example, if there are columns *governor* and *mayor*, then the ATTRIBUTE...DETERMINES statement would be state to governor and city to mayor.

In our example, suppose a query was issued that queried by *mayor* instead of *city*. Since this 1-1 relationship exists between the attribute and the level, city can be used to identify the data.

```
ATTRIBUTE     city      DETERMINES   mayor
```

This complete dimension definition is shown below including the creation of the location table.

```
CREATE TABLE location  (
       city     VARCHAR2(30),
       state    VARCHAR2(30),
       country  VARCHAR2(30),
       mayor    VARCHAR2(30),
       governor  VARCHAR2(30)  );

CREATE DIMENSION location_dim
       LEVEL city      IS location.city
       LEVEL state     IS location.state
       LEVEL country   IS location.country
HIERARCHY loc_rollup    (
       city     CHILD OF
       state    CHILD OF
       country )
ATTRIBUTE     city      DETERMINES   location.mayor
ATTRIBUTE     state     DETERMINES   location.governor;
```

The design, creation, and maintenance of dimensions is part of the design, creation, and maintenance of your data warehouse schema. Once the dimension has been created, check that it meets these requirements:

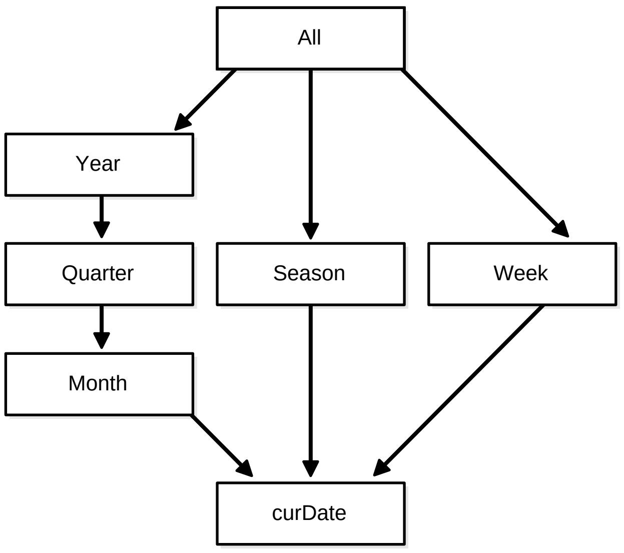
- **There must be a 1:n relationship between a parent and children.** A parent can have one or more children, but a child can have only one parent.
- **There must be a 1:1 attribute relationship between hierarchy levels and their dependent dimension attributes.** For example, if there is a column *corporation*, then a possible attribute relationship would be *corporation* to *president*.
- **If the columns of a parent level and child level are in different relations, then the connection between them also requires a 1:n join relationship.** Each row of the child table must join with one and only one row of the parent table. This relationship is stronger than referential integrity alone because it requires that the child join key must be non-null, that referential integrity must be maintained from the child join key to the parent join key, and that the parent join key must be unique.

- **Ensure (using database constraints if necessary) that the columns of each hierarchy level are non-null and that hierarchical integrity is maintained.**
- **The hierarchies of a dimension may overlap or be disconnected from each other.** However, the columns of a hierarchy level cannot be associated with more than one dimension.
- **Join relationships that form cycles in the dimension graph are not supported.** For example, a hierarchy level cannot be joined to itself either directly or indirectly.

Multiple Hierarchies

A single dimension definition can contain multiple hierarchies as illustrated below. Suppose a department store wants to track the sales of certain items over time. The first step is to define the time dimension over which sales will be tracked. Figure 30–2 on page 30-6 illustrates a dimension "Time" with three time hierarchies.

Figure 30–2 Time_dim Dimension with Three Time Hierarchies



From the illustration, you can construct the following denormalized Time dimension statement. The associated CREATE TABLE statement is also shown.

```
CREATE TABLE time (
        curDate      DATE,
        month        INTEGER,
        quarter      INTEGER,
        year         INTEGER,
        season       INTEGER,
        week_num     INTEGER,
        dayofweek    VARCHAR2(30),
        month_name   VARCHAR2(30)  );

CREATE DIMENSION time_dim
    LEVEL curDate     IS time.curDate
    LEVEL month       IS time.month
    LEVEL quarter     IS time.quarter
    LEVEL year        IS time.year
    LEVEL season      IS time.season
    LEVEL week_num    IS time.week_num

HIERARCHY calendar_rollup  (
    curDate       CHILD OF
    month         CHILD OF
    quarter       CHILD OF
    year                    )
HIERARCHY weekly_rollup    (
         curDate          CHILD OF
         week_num           )
HIERARCHY seasonal_rollup  (
         curDate          CHILD OF
         season             )
ATTRIBUTE curDate     DETERMINES  time.dayofweek
ATTRIBUTE month       DETERMINES  time.month_name;
```

Using Normalized Dimension Tables

The tables used to define a dimension may be normalized or denormalized and the individual hierarchies can be normalized or denormalized. If the levels of a hierarchy come from the same table, it is called a fully denormalized hierarchy. For example, CALENDAR_ROLLUP in the Time dimension is a denormalized hierarchy. If levels of a hierarchy come from different tables, such a hierarchy is either a fully or partially normalized hierarchy. This section shows how to define a normalized hierarchy.

Suppose the tracking of products is done by product, brand, and department. This data is stored in the tables PRODUCT, BRAND, and DEPARTMENT. The product dimension is *normalized* because the data entities ITEM_NAME, BRAND_ID, and DEPT_ID are taken from different tables. The clause JOIN KEY within the dimension definition specifies how to join together the levels in the hierarchy. The dimension statement and the associated CREATE TABLE statements for the PRODUCT, BRAND, and DEPARTMENT tables are shown below.

```
CREATE TABLE product  (
        item_name    VARCHAR2(30),
        brand_id     INTEGER  );

CREATE TABLE brand  (
        brand_id     INTEGER,
        brand_name   VARCHAR2(30),
        dept_id      INTEGER);

CREATE TABLE department  (
        dept_id       INTEGER,
        dept_name     VARCHAR2(30),
        dept_type     INTEGER);

CREATE DIMENSION product_dim
    LEVEL item      IS product.item_name
    LEVEL brand_id  IS brand.brand_id
    LEVEL dept_id   IS department.dept_id

HIERARCHY merchandise_rollup
(
    item                   CHILD OF
    brand_id               CHILD OF
    dept_id

    JOIN KEY  product.brand_id REFERENCES brand_id
    JOIN KEY  brand.dept_id    REFERENCES dept_id
)
ATTRIBUTE brand_id DETERMINES product.brand_name
ATTRIBUTE dept_id  DETERMINES (product.dept_name, product.dept_type);
```

Viewing Dimensions

Two procedures are available which allow you to display the dimensions that have been defined. First, the file smdim.sql must be executed to provide the DEMO_DIM package, which includes:

- DEMO_DIM.PRINT_DIM to print a specific dimension

- DEMO_DIM.PRINT_ALLDIMS to print all dimensions

The DEMO_DIM.PRINT_DIM procedure has only one parameter, the name of the dimension to display. The example below shows how to display the dimension TIME_PD.

```
DEMO_DIM.PRINT_DIM  ('TIME_PD');
```

To display all of the dimensions that have been defined, call the procedure DEMO_DIM.PRINT_ALLDIMS without any parameters as shown below.

```
DEMO_DIM.PRINT_ALLDIMS ();
```

Irrespective of which procedure is called, the output is identical. A sample display is shown below.

```
DIMENSION GROCERY.TIME_PD
LEVEL FISCAL_QTR IS GROCERY.WEEK.FISCAL_QTR
LEVEL MONTH IS GROCERY.MONTH.MONTH
LEVEL QUARTER IS GROCERY.QUARTER.QUARTER
LEVEL TIME_KEY IS GROCERY.TIME.TIME_KEY
LEVEL WEEK IS GROCERY.WEEK.WEEK
LEVEL YEAR IS GROCERY.YEAR.YEAR
HIERARCHY WEEKLY_ROLLUP (
      TIME_KEY
       CHILD OF WEEK
   JOIN KEY GROCERY.TIME.WEEK REFERENCES WEEK
  )
    HIERARCHY FISCAL_ROLLUP (
             TIME_KEY
             CHILD OF WEEK
             CHILD OF FISCAL_QTR
    JOIN KEY GROCERY.TIME.WEEK REFERENCES WEEK
    )
   HIERARCHY CALENDAR_ROLLUP (
             TIME_KEY
             CHILD OF MONTH
             CHILD OF QUARTER
             CHILD OF YEAR
      JOIN KEY GROCERY.TIME.MONTH REFERENCES MONTH
      JOIN KEY GROCERY.MONTH.QUARTER REFERENCES QUARTER
      JOIN KEY GROCERY.QUARTER.YEAR REFERENCES YEAR
    )

    ATTRIBUTE TIME_KEY DETERMINES GROCERY.TIME.DAY_NUMBER_IN_MONTH
    ATTRIBUTE TIME_KEY DETERMINES GROCERY.TIME.DAY_NUMBER_IN_YEAR
    ATTRIBUTE WEEK DETERMINES GROCERY.WEEK.WEEK_NUMBER_OF_YEAR
    ATTRIBUTE MONTH DETERMINES GROCERY.MONTH.FULL_MONTH_NAME
```

Dimensions and Constraints

Constraints plan an important role with dimensions. In most cases, full referential integrity is enforced on the operational databases, and operational procedures can be used to ensure that data flowing into the data warehouse (after data cleansing) never violates referential integrity; so, in practice, referential integrity constraints may or may not be enabled in the data warehouse.

It is recommended that constraints be enabled and, if validation time is a concern, then the NOVALIDATE clause should be used as shown below. Primary and foreign keys should be implemented as described. Referential integrity constraints and `NOT NULL` constraints on the fact tables provide information that query rewrite can use to extend the usefulness of materialized views.

```
ENABLE NOVALIDATE CONSTRAINT pk_time;
```

In addition, the RELY clause should be used to advise query rewrite that it can rely upon the constraints being correct as shown below.

```
ALTER TABLE time MODIFY CONSTRAINT pk_time RELY;
```

Validating a Dimension

If the relationships described by the dimensions are incorrect, wrong results could occur. Therefore, you should verify the relationships specified by CREATE DIMENSION using the DBMS_OLAP.VALIDATE_DIMENSION procedure periodically.

This procedure is easy to use and only has four parameters:

- dimension name
- owner name
- set to TRUE to only check the new rows for tables of this dimension
- set to TRUE to verify that all columns are not null

The example shown below validates the dimension time_fn in the Grocery schema

```
DBMS_OLAP.VALIDATE_DIMENSION ('TIME_FN', 'GROCERY', FALSE, TRUE);
```

All exceptions encountered by the VALIDATE_DIMENSION procedure are placed in the table MVIEW$_EXCEPTIONS, which is created in the user's schema. Querying this table will identify the exceptions that were found. For example:

```
OWNER    TABLE_NAME  DIMENSION_NAME RELATIONSHIP BAD_ROWID
-------- ----------- -------------- ------------ ------------------
GROCERY  MONTH       TIME_FN        FOREIGN KEY  AAAAuwAAJAAAARwAAA
```

However, rather than query this table, it may be better to query as follows where the rowid of the invalid row is used to retrieve the actual row that has violated the constraint. In this example, the dimension TIME_FD is checking a table called month. It has found a row that violates the constraints and using the rowid, we can see exactly which row in the month table is causing the problem.

```
SELECT * FROM month
WHERE rowid IN (select bad_rowid from mview$_exceptions);

MONTH      QUARTER    FISCAL_QTR YEAR       FULL_MONTH_NAME      MONTH_NUMB
---------- ---------- ---------- ---------- -------------------- ----------
199903     19981      19981      1998       March                3
```

Altering a Dimension

Some modification can be made to the dimension using the ALTER DIMENSION statement. You can add or drop a level, hierarchy, or attribute from the dimension using this command.

Referring to the time dimension in Figure 30–2, you could remove the attribute month, drop the hierarchy weekly_rollup, and remove the level week. In addition, you could add a new level called qtr1.

```
ALTER DIMENSION time_dim DROP attribute month;
ALTER DIMENSION time_dim DROP hierarchy weekly_rollup;
ALTER DIMENSION time_dim DROP LEVEL week;
ALTER DIMENSION time_dim ADD LEVEL qtr1 IS time.fiscal_qtr;
```

A dimension becomes invalid if you change any schema object which the dimension is referencing. For example, if the table on which the dimension is defined is altered.

To check the status of a dimension, view the contents of the column *invalid* in the table ALL_DIMENSIONS.

To revalidate the dimension, use the COMPILE option as shown below.

```
ALTER DIMENSION time_dim COMPILE;
```

Deleting a Dimension

A dimension is removed using the DROP DIMENSION command. For example:

```
DROP DIMENSION time_dim;
```

31

Query Rewrite

This chapter contains:

Overview of Query Rewrite

One of the major benefits of creating and maintaining materialized views is the ability to take advantage of query rewrite, which transforms a SQL statement expressed in terms of tables or views into a statement accessing one or more materialized views that are defined on the detail tables. The transformation is transparent to the end user or application, requiring no intervention and no reference to the materialized view in the SQL statement. Because query rewrite is transparent, materialized views can be added or dropped just like indexes without invalidating the SQL in the application code.

Before the query is rewritten, it is subjected to several checks to determine whether it is a candidate for query rewrite. If the query fails any of the checks, then the query is applied to the detail tables rather than the materialized view. This can be costly in terms of response time and processing power.

The Oracle optimizer uses two different methods to recognize when to rewrite a query in terms of one or more materialized views. The first method is based on matching the SQL text of the query with the SQL text of the materialized view definition. If the first method fails, the optimizer uses the more general method in which it compares join conditions, data columns, grouping columns, and aggregate functions between the query and a materialized view.

Query rewrite operates on queries and subqueries in the following types of SQL statements:

- SELECT
- CREATE TABLE … AS SELECT
- INSERT INTO … SELECT

and on subqueries in the set operators UNION, UNION ALL, INTERSECT, and MINUS.

Several factors affect whether or not a given query is rewritten to use one or more materialized views:

- Enabling / disabling query rewrite
  - by the CREATE or ALTER statement for individual materialized views
  - by the initialization parameter QUERY_REWRITE_ENABLED
  - by the REWRITE and NOREWRITE hints in SQL statements
- Rewrite integrity levels
- Dimensions and constraints

Cost-Based Rewrite

Query rewrite is available with cost-based optimization. Oracle optimizes the input query with and without rewrite and selects the least costly alternative. The optimizer rewrites a query by rewriting one or more query blocks, one at a time.

If the rewrite logic has a choice between multiple materialized views to rewrite a query block, it will select one to optimize the ratio of the sum of the cardinality of the tables in the rewritten query block to that in the original query block. Therefore, the materialized view selected would be the one which can result in reading in the least amount of data.

After a materialized view has been picked for a rewrite, the optimizer performs the rewrite, and then tests whether the rewritten query can be rewritten further with another materialized view. This process continues until no further rewrites are possible. Then the rewritten query is optimized and the original query is optimized. The optimizer compares these two optimizations and selects the least costly alternative.

Since optimization is based on cost, it is important to collect statistics both on tables involved in the query and on the tables representing materialized views. Statistics are fundamental measures, such as the number of rows in a table, that are used to calculate the cost of a (rewritten) query. They are created with the ANALYZE statement or by using the DBMS_STATISTICS package.

Queries that contain in-line or named views are also candidates for query rewrite. When a query contains a named view, the view name is used to do the matching between a materialized view and the query. That is, the set of named views in a materialized view definition should match exactly with the set of views in the query. When a query contains an inline view, the inline view may be merged into the query before matching between a materialized view and the query occurs.

Enabling Query Rewrite

Several steps must be followed to enable query rewrite:

1. Individual materialized views must have the ENABLE QUERY REWRITE clause.
2. The initialization parameter QUERY_REWRITE_ENABLED must be set to TRUE.
3. Cost-based optimization must be used either by setting the initialization parameter OPTIMIZER_MODE to "ALL_ROWS" or "FIRST_ROWS", or by analyzing the tables and setting OPTIMIZER_MODE to "CHOOSE".

If step 1 has not been completed, a materialized view will never be eligible for query rewrite. ENABLE QUERY REWRITE can be specified either when the materialized view is created, as illustrated below, or via the ALTER MATERIALIZED VIEW statement.

```
CREATE MATERIALIZED VIEW store_sales_mv
 ENABLE QUERY REWRITE
 AS
 SELECT s.store_name,
   SUM(dollar_sales) AS sum_dollar_sales
 FROM store s,  fact f
```

```
WHERE f.store_key = s.store_key
GROUP BY s.store_name;
```

You can use the initialization parameter QUERY_REWRITE_ENABLED to disable query rewrite for all materialized views, or to enable it again for all materialized views that are individually enabled. However, the QUERY_REWRITE_ENABLED parameter cannot enable query rewrite for materialized views that have disabled it with the CREATE or ALTER statement.

The NOREWRITE hint disables query rewrite in a SQL statement, overriding the QUERY_REWRITE_ENABLED parameter, and the REWRITE (*mview_name*, ...) hint restricts the eligible materialized views to those named in the hint.

Initialization Parameters for Query Rewrite

Query rewrite requires the following initialization parameter settings:

- OPTIMIZER_MODE = "ALL_ROWS", "FIRST_ROWS", or "CHOOSE"
- QUERY_REWRITE_ENABLED = TRUE
- COMPATIBLE = 8.1.0 (or greater)

The QUERY_REWRITE_INTEGRITY parameter is optional, but must be set to STALE_TOLERATED, TRUSTED, or ENFORCED if it is specified (see "Accuracy of Query Rewrite" on page 31-18). It will default to ENFORCED if it is undefined.

Privileges for Enabling Query Rewrite

A materialized view is used based not on privileges the user has on that materialized view, but based on privileges the user has on detail tables or views in the query.

The system privilege GRANT REWRITE allows you to enable materialized views in your own schema for query rewrite only if all tables directly referenced by the materialized view are in that schema. The GRANT GLOBAL REWRITE privilege allows you to enable materialized views for query rewrite even if the materialized view references objects in other schemas.

The privileges for using materialized views for query rewrite are similar to those for definer-rights procedures. See *Oracle8i Concepts* for further information.

When Does Oracle Rewrite a Query?

A query gets rewritten only when a certain number of conditions are met:

1. Query rewrite must be enabled for the session.
2. A materialized view must be enabled for query rewrite.
3. The rewrite integrity level should allow the use of the materialized view. For example, if a materialized view is not fresh and query rewrite integrity is set to ENFORCED, then the materialized view will not be used.
4. Either all or part of the results requested by the query must be obtainable from the precomputed result stored in the materialized view.

To determine this, the optimizer may depend on some of the data relationships declared by the user via constraints and dimensions. Such data relationships include hierarchies, referential integrity, and uniqueness of key data, and so on.

The following sections use an example schema and a few materialized views to illustrate how the data relationships are used by the optimizer to rewrite queries. A retail database consists of these tables:

```
STORE   (store_key, store_name, store_city, store_state, store_country)
PRODUCT (prod_key, prod_name, prod_brand)
TIME    (time_key, time_day, time_week, time_month)
FACT    (store_key, prod_key, time_key, dollar_sales)
```

Two materialized views created on these tables contain only joins:

```
CREATE MATERIALIZED VIEW join_fact_store_time
  ENABLE QUERY REWRITE
  AS
  SELECT s.store_key, s.store_name, f.dollar_sales, t.time_key, t.time_day,
         f.prod_key, f.rowid, t.rowid
  FROM   fact f, store s, time t
  WHERE  f.time_key = t.time_key AND f.store_key = s.store_key;

CREATE MATERIALIZED VIEW join_fact_store_time_oj
  ENABLE QUERY REWRITE
  AS
  SELECT s.store_key, s.store_name, f.dollar_sales, t.time_key,
         f.rowid, t.rowid
  FROM   fact f, store s, time t
  WHERE  f.time_key = t.time_key(+) AND f.store_key = s.store_key(+);
```

and two materialized views contain joins and aggregates:

```
CREATE MATERIALIZED VIEW sum_fact_store_time_prod
  ENABLE QUERY REWRITE
  AS
  SELECT  s.store_name, time_week, p.prod_key,
          SUM(f.dollar_sales) AS sum_sales,
          COUNT(f.dollar_sales) AS count_sales
  FROM   fact f, store s, time t, product p
  WHERE  f.time_key = t.time_key  AND  f.store_key = s.store_key AND
         f.prod_key = p.prod_key
  GROUP BY s.store_name, time_week, p.prod_key;

CREATE MATERIALIZED VIEW sum_fact_store_prod
  ENABLE QUERY REWRITE
  AS
  SELECT  s.store_city, p.prod_name
          SUM(f.dollar_sales) AS sum_sales,
          COUNT(f.dollar_sales) AS count_sales
  FROM   fact f, store s, product p
  WHERE   f.store_key = s.store_key  AND f.prod_key = p.prod_key
  GROUP BY store_city, p.prod_name;
```

You must compute statistics on the materialized views so that the optimizer can determine based on cost whether to rewrite the queries.

```
ANALYZE TABLE join_fact_store_time COMPUTE STATISTICS;
ANALYZE TABLE join_fact_store_time_oj COMPUTE STATISTICS;
ANALYZE TABLE sum_fact_store_time_prod COMPUTE STATISTICS;
ANALYZE TABLE sum_fact_store_prod COMPUTE STATISTICS;
```

Query Rewrite Methods

The optimizer uses a number of different methods to rewrite a query. The first, most important step is to determine if all or part of the results requested by the query can be obtained from the precomputed results stored in a materialized view.

The simplest case occurs when the result stored in a materialized view exactly matches what is requested by a query. The Oracle optimizer makes this type of determination by comparing the SQL text of the query with the SQL text of the materialized view definition. This method is most straightforward and also very limiting.

When the SQL text comparison test fails, the Oracle optimizer performs a series of generalized checks based on the joins, grouping, aggregates, and column data fetched. This is accomplished by individually comparing various clauses (SELECT, FROM, WHERE, GROUP BY) of a query with those of a materialized view.

SQL Text Match Rewrite Methods

Two methods are used by the optimizer:

1. Full SQL text match
2. Partial SQL text match

In full SQL text match, the entire SQL text of a query is compared against the entire SQL text of a materialized view definition (that is, the entire SELECT expression), ignoring the white space during SQL text comparison. The following query

```
SELECT s.store_name, time_week, p.prod_key,
       SUM(f.dollar_sales) AS sum_sales,
       COUNT(f.dollar_sales) AS count_sales
FROM   fact f, store s, time t, product p
WHERE  f.time_key = t.time_key  AND
       f.store_key = s.store_key AND
       f.prod_key = p.prod_key
GROUP BY s.store_name, time_week, p.prod_key;
```

which matches *sum_fact_store_time_prod* (white space excluded) will be rewritten as:

```
SELECT store_name, time_week, product_key, sum_sales, count_sales
FROM   sum_fact_store_time_prod;
```

When full SQL text match fails, the optimizer then attempts a partial SQL text match. In this method, the SQL text starting from the FROM clause of a query is compared against the SQL text starting from the FROM clause of a materialized view definition. Therefore, this query:

```
SELECT s.store_name, time_week, p.prod_key,
       AVG(f.dollar_sales) AS avg_sales
FROM   fact f, store s, time t, product p
WHERE  f.time_key = t.time_key  AND
       f.store_key = s.store_key AND
       f.prod_key = p.prod_key
GROUP BY s.store_name, time_week, p.prod_key;
```

will be rewritten as:

```
SELECT store_name, time_week, prod_key, sum_sales/count_sales AS avg_sales
FROM   sum_fact_store_time_prod;
```

Note that under the partial SQL text match rewrite method, the average of sales aggregate required by the query is computed using sum of sales and count of sales aggregates stored in the materialized view.

When neither SQL text match succeeds, the optimizer uses a general query rewrite method.

General Query Rewrite Methods

The general query rewrite methods are much more powerful than SQL text match methods because they can enable the use of a materialized view even if it contains only part of the data requested by a query, or it contains more data than what is requested by a query, or it contains data in a different form which can be converted into a form required by a query. To achieve this, the optimizer compares the SQL clauses (SELECT, FROM, WHERE, GROUP BY) individually between a query and a materialized view.

The Oracle optimizer employs four different checks called:

- Join Compatibility
- Data Sufficiency
- Grouping Compatibility
- Aggregate Computability

Depending on the type of a materialized view, some or all four checks are made to determine if the materialized view can be used to rewrite a query as illustrated in the table below.

Table 31–1 Materialized View Types and General Query Rewrite Methods

	MV with Joins Only	MV with Joins and Aggregates	MV with Aggregates on a Single Table
Join Compatibility	X	X	-
Data Sufficiency	X	X	X
Grouping Compatibility	-	X	X
Aggregate Computability	-	X	X

To perform these checks, the optimizer uses data relationships on which it can depend. For example, primary key and foreign key relationships tell the optimizer that each row in the foreign key table joins with at most one row in the primary key table. Furthermore, if there is a NOT NULL constraint on the foreign key, it indicates that each row in the foreign key table joins with exactly one row in the primary key table.

Data relationships such as these are very important for query rewrite because they tell what type of result is produced by joins, grouping, or aggregation of data. Therefore, to maximize the rewritability of a large set of queries when such data relationships exist in a database, they should be declared by the user.

Join Compatibility Check

In this check, the joins in a query are compared against the joins in a materialized view. In general, this comparison results in the classification of joins into three categories:

1. Common joins that occur in both the query and the materialized view
2. Delta joins that occur in the query but not in the materialized view
3. Delta joins that occur in the materialized view but not in the query

Common Joins The common join pairs between the two must be of same type, or the join in the query must be derivable from the join in the materialized view. For example, if a materialized view contains an outer join of table A with table B, and a query contains an inner join of table A with table B, the result of the inner join can be derived by filtering the anti-join rows from the result of the outer join.

For example, consider this query:

```
SELECT s.store_name, t.time_day, SUM(f.dollar_sales)
FROM   fact f, store s, time t
WHERE  f.time_key = t.time_key AND
       f.store_key = s.store_key AND
       t.time_day BETWEEN '01-DEC-1997' AND '31-DEC-1997'
GROUP BY s.store_name, t.time_day;
```

The common joins between this query and the materialized view *join_fact_store_time* are:

```
f.time_key = t.time_key AND  f.store_key = s.store_key
```

They match exactly and the query can be rewritten as:

```
SELECT store_name, time_day, SUM(dollar_sales)
FROM   join_fact_store_time
WHERE  time_day BETWEEN '01-DEC-1997' AND '31-DEC-1997'
GROUP BY store_name, time_day;
```

The query could also be answered using the *join_fact_store_time_oj* materialized view where inner joins in the query can be derived from outer joins in the materialized view. The rewritten version will (transparently to the user) filter out the anti-join rows. The rewritten query will have the structure:

```
SELECT store_name, time_day, SUM(f.dollar_sales)
FROM   join_fact_store_time_oj
WHERE  time_key IS NOT NULL AND store_key IS NOT NULL AND
       time_day BETWEEN '01-DEC-1997' AND '31-DEC-1997'
GROUP BY store_name, time_day;
```

In general, if you use an outer join in a materialized view containing only joins, you should put in the materialized view either the primary key or the rowid on the right side of the outer join. For example, in the previous example, *join_fact_store_time_oj* there is a primary key on both store and time.

Another example of when a materialized view containing only joins is used is the case of a semi-join rewrites. That is, a query contains either an EXISTS or an IN subquery with a single table.

Consider this query, which reports the stores that had sales greater than $10,000 during the 1997 Christmas season.

```
SELECT DISTINCT store_name
FROM store s
WHERE EXISTS (SELECT *
              FROM fact f
              WHERE f.store_key = s.store_key
                AND f.dollar_sales > 10000
                and f.time_key between '01-DEC-1997' and '31-DEC-1997');
```

This query could also be seen as:

```
SELECT DISTINCT store_name
FROM store s
WHERE s.store_key in (SELECT f.store_key
                     FROM fact f
                     WHERE f.dollar_sales > 10000);
```

This query contains a semi-join 'f.store_key = s.store_key' between the store and the fact table. This query can be rewritten to use either the *join_fact_store_time* materialized view, if foreign key constraints are active or *join_fact_store_time_oj* materialized view, if primary keys are active. Observe that both materialized views contain 'f.store_key = s.store_key' which can be used to derive the semi-join in the query.

The query is rewritten with *join_fact_store_time* as follows:

```
SELECT store_name
FROM (SELECT DISTINCT store_name, store_key
      FROM join_fact_store_time
      WHERE dollar_sales > 10000
      AND f.time_key BETWEEN ’01-DEC-1997’ and ’31-DEC-1997’);
```

If the materialized view *join_fact_store_time* is partitioned by *time_key*, then this query is likely to be more efficient than the original query because the original join between *store* and *fact* has been avoided.

The query could be rewritten using *join_fact_store_time_oj* as follows.

```
SELECT store_name
FROM (SELECT DISTINCT store_name, store_key
      FROM join_fact_store_time_oj
      WHERE dollar_sales > 10000
        AND store_key IS NOT NULL
        AND time_key BETWEEN ’01-DEC-1997’ and ’31-DEC-1997’);
```

Rewrites with semi-joins are currently restricted to materialized views with joins only and are not available for materialized views with joins and aggregates.

Query Delta Joins A *query delta join* is a join that appears in the query but not in the materialized view. Any number and type of delta joins in a query are allowed and they are simply retained when the query is rewritten with a materialized view. Upon rewrite, the materialized view is joined to the appropriate tables in the delta joins.

For example, consider this query:

```
SELECT store_name, prod_name, SUM(f.dollar_sales)
FROM   fact f, store s, time t, product p
WHERE  f.time_key = t.time_key AND
       f.store_key = s.store_key AND
       f.prod_key = p.prod_key AND
       t.time_day BETWEEN ’01-DEC-1997’ AND ’31-DEC-1997’
GROUP BY  store_name, prod_name;
```

Using the materialized view *join_fact_store_time*, common joins are: f.time_key = t.time_key AND f.store_key = s.store_key. The delta join in the query is f.prod_key = p.prod_key.

The rewritten form will then join the *join_fact_store_time* materialized view with the *product* table:

```
SELECT store_name, prod_name, SUM(f.dollar_sales)
FROM   join_fact_store_time mv,  product p
WHERE  mv.prod_key = p.prod_key AND
       mv.time_day BETWEEN '01-DEC-1997' AND '31-DEC-1997'
GROUP BY store_name, prod_name;
```

Materialized View Delta Joins All delta joins in a materialized view are required to be lossless with respect to the result of common joins. A lossless join guarantees that the result of common joins is not restricted. A *lossless* join is one where, if two tables called A and B are joined together, rows in table A will always match with rows in table B and no data will be lost, hence the term lossless join. For example, every row with the foreign key matches a row with a primary key provided no nulls are allowed in the foreign key. Therefore, to guarantee a lossless join, it is necessary to have FOREIGN KEY, PRIMARY KEY, and NOT NULL constraints on appropriate join keys. Alternatively, if the join between tables A and B is an outer join (A being the outer table), it is lossless as it preserves all rows of table A.

All delta joins in a materialized view are required to be non-duplicating with respect to the result of common joins. A non-duplicating join guarantees that the result of common joins is not duplicated. For example, a non-duplicating join is one where, if table A and table B are joined together, rows in table A will match with at most one row in table B and no duplication occurs. To guarantee a non-duplicating join, the key in table B must be constrained to unique values by using a primary key or unique constraint.

Consider this query which joins FACT and TIME:

```
SELECT t.time_day, sum(f.dollar_sales)
FROM   fact f, time t
WHERE  f.time_key = t.time_key AND
       t.time_day BETWEEN '01-DEC-1997' AND '31-DEC-1997'
GROUP  t.time_day;
```

The materialized view *join_fact_store_time* has an additional join between FACT and STORE: 'f.store_key = s.store_key'. This is the delta join in *join_fact_store_time*.

We can rewrite the query if this join is lossless and non-duplicating. This is the case if f.store_key is a foreign key to p.store_key and is not null. The query is therefore rewritten as:

```
SELECT time_day, SUM(f.dollar_sales)
FROM   join_fact_store_time
WHERE  time_day BETWEEN '01-DEC-1997' AND '31-DEC-1997'
GROUP BY time_day;
```

The query could also be rewritten with the materialized view *join_fact_store_time_oj* where foreign key constraints are not needed. This view contains an outer join between fact and store: ’f.store_key = s.store_key(+)’ which makes the join lossless. If s.store_key is a primary key, then the non-duplicating condition is satisfied as well and optimizer will rewrite the query as:

```
SELECT time_day, SUM(f.dollar_sales)
FROM   join_fact_store_time_oj
WHERE  time_key IS NOT NULL AND
       time_day BETWEEN ’01-DEC-1997’ AND ’31-DEC-1997’
GROUP BY time_day;
```

The current limitations restrict most rewrites with outer joins to materialized views with joins only. There is very limited support for rewrites with materialized aggregate views with outer joins. Those views should rely on foreign key constraints to assure losslessness of delta materialized view joins.

Data Sufficiency Check

In this check, the optimizer determines if the necessary column data requested by a query can be obtained from a materialized view. For this, the equivalence of one column with another is used. For example, if an inner join between table A and table B is based on a join predicate A.X = B.X, then the data in column A.X will equal the data in column B.X in the result of the join. This data property is used to match column A.X in a query with column B.X in a materialized view or vice versa.

For example, consider this query:

```
SELECT s.store_name, f.time_key, SUM(f.dollar_sales)
FROM   fact f, store s, time t
WHERE  f.time_key = t.time_key AND
       f.store_key = s.store_key
GROUP BY s.store_name, f.time_key;
```

This query can be answered with *join_fact_store_time* even though the materialized view doesn’t have f.time_key. Instead, it has t.time_key which, through a join condition ’f.time_key = t.time_key’, is equivalent to f.time_key.

Thus, the optimizer may select this rewrite:

```
SELECT store_name, time_day, SUM(dollar_sales)
FROM   join_fact_store_time
GROUP BY store_name, time_key;
```

If some column data requested by a query cannot be obtained from a materialized view, the optimizer further determines if it can be obtained based on a data relationship called functional dependency. When the data in a column can determine data in another column, such a relationship is called functional dependency or functional determinance. For example, if a table contains a primary key column called *prod_key* and another column called *prod_name*, then, given a *prod_key* value, it is possible to look up the corresponding *prod_name*. The opposite is not true, which means a *prod_name* value need not relate to a unique *prod_key*.

When the column data required by a query is not available from a materialized view, such column data can still be obtained by joining the materialized view back to the table that contains required column data provided the materialized view contains a key that functionally determines the required column data.

For example, consider this query:

```
SELECT  s.store_name, t.time_week, p.prod_name,
        SUM(f.dollar_sales) AS sum_sales,
FROM   fact f, store s, time t, product p
WHERE  f.time_key = t.time_key  AND  f.store_key = s.store_key AND
       f.prod_key = p.prod_key  AND p.prod_brand = 'KELLOGG'
GROUP BY s.store_name, t.time_week, p.prod_name;
```

The materialized view *sum_fact_store_time_prod* contains *p.prod_key*, but not *p.prod_brand*. However, we can join *sum_fact_store_time_prod* back to PRODUCT to retrieve *prod_brand* because *prod_key* functionally determines *prod_brand*. The optimizer rewrites this query using *sum_fact_store_time_prod* as:

```
SELECT  mv.store_name, mv.time_week, p.product_key, mv.sum_sales,
FROM    sum_fact_store_time_prod mv, product p
WHERE   mv.prod_key = p.prod_key  AND p.prod_brand = 'KELLOGG'
GROUP BY mv.store_name, mv.time_week, p.prod_key;
```

Here the PRODUCT table is called a joinback table because it was originally joined in the materialized view but joined back again in the rewritten query.

There are two ways to declare functional dependency:

1. Using the primary key constraint
2. Using the DETERMINES clause of a dimension

The DETERMINES clause of a dimension definition may be the only way you could declare functional dependency when the column that determines another column cannot be a primary key. For example, the STORE table is a denormalized dimension table which has columns *store_key*, *store_name*, *store_city*, *city_name*, and *store_state*. *Store_key* functionally determines *store_name* and *store_city* functionally determines *store_state*.

The first functional dependency can be established by declaring store_key as the primary key, but not the second functional dependency because the store_city column contains duplicate values. In this situation, you can use the DETERMINES clause of a dimension to declare the second functional dependency.

The following dimension definition illustrates how the functional dependencies are declared.

```
CREATE DIMENSION store_dim
LEVEL store_key      IS store.store_key
LEVEL city           IS store.store_city
LEVEL state          IS store.store_state
LEVEL country        IS store.store_country
  HIERARCHY geographical_rollup          (
             store_key     CHILD OF
             city          CHILD OF
             state         CHILD OF
             country                   )
ATTRIBUTE store_key DETERMINES store.store_name;
ATTRIBUTE store_city DETERMINES store.city_name;
```

The hierarchy *geographic_rollup* declares hierarchical relationships which are also 1:n functional dependencies. The 1:1 functional dependencies are declared using the DETERMINES clause, such as store_city functionally determines city_name.

The following query:

```
SELECT  s.store_city, p.prod_name
          SUM(f.dollar_sales) AS sum_sales,
FROM    fact f, store s, product p
WHERE   f.store_key = s.store_key AND f.prod_key = p.prod_key
          AND s.city_name = 'BELMONT'
GROUP BY s.store_city, p.prod_name;
```

can be rewritten by joining *sum_fact_store_prod* to the STORE table so that *city_name* is available to evaluate the predicate. But the join will be based on the *store_city* column, which is not a primary key in the STORE table; therefore, it allows duplicates. This is accomplished by using an inline view which selects distinct values and this view is joined to the materialized view as shown in the rewritten query below.

```
SELECT  iv.store_city, mv.prod_name, mv.sum_sales
FROM    sum_fact_store_prod mv, (SELECT DISTINCT store_city, city_name
                                   FROM store) iv
WHERE   mv.store_city = iv.store_city AND
          iv.store_name = 'BELMONT'
GROUP BY iv.store_city, mv.prod_name;
```

This type of rewrite was possible because the fact that *store_city* functionally determines *city_name* as declared in the dimension.

Grouping Compatibility Check

This check is required only if both the materialized view and the query contain a GROUP BY clause. The optimizer first determines if the grouping of data requested by a query is exactly the same as the grouping of data stored in a materialized view. That means, the level of grouping is the same in both the query and the materialized view. For example, a query requests data grouped by *store_city* and a materialized view stores data grouped by *store_city* and *store_state*. The grouping is the same in both provided *store_city* functionally determines *store_state,* such as the functional dependency shown in the dimension example above.

If the grouping of data requested by a query is at a coarser level compared to the grouping of data stored in a materialized view, the optimizer can still use the materialized view to rewrite the query. For example, the materialized view *sum_fact_store_time_prod* groups by *store_name*, *time_week*, and *prod_key*. This query groups by *store_name*, a coarser grouping granularity:

```
SELECT s.store_name, SUM(f.dollar_sales) AS sum_sales,
FROM   fact f, store s
WHERE  f.store_key = s.store_key
GROUP BY s.store_name;
```

Therefore, the optimizer will rewrite this query as:

```
SELECT store_name, SUM(sum_dollar_sales) AS sum_sales,
FROM   sum_fact_store_time_prod
GROUP BY s.store_name;
```

In another example, a query requests data grouped by *store_state* whereas a materialized view stores data grouped by *store_city*. If *store_city* is a CHILD OF *store_state* (see the dimension example above), the grouped data stored in the materialized view can be further grouped by *store_state* when the query is rewritten. In other words, aggregates at *store_city* level (finer granularity) stored in a materialized view can be rolled up into aggregates at *store_state* level (coarser granularity).

For example, consider the following query:

```
SELECT  store_state, prod_name, SUM(f.dollar_sales) AS sum_sales
FROM    fact f, store s, product p
WHERE   f.store_key = s.store_key  AND  f.prod_key = p.prod_key
GROUP BY store_state, prod_name;
```

Because store_city functionally determines *store_state*, *sum_fact_store_prod* can be used with a joinback to store table to retrieve *store_state* column data, and then aggregates can be rolled up to *store_state* level, as shown below:

```
SELECT  store_state, prod_name, sum(mv.sum_sales) AS sum_sales
FROM    sum_fact_store_prod mv, (SELECT DISTINCT store_city, store_state
                                 FROM store) iv
WHERE   mv.store_city = iv.store_city
GROUP BY store_state, prod_name;
```

Note that for this rewrite, the data sufficiency check determines that a joinback to the STORE table is necessary, and the grouping compatibility check determines that aggregate rollup is necessary.

Aggregate Computability Check

This check is required only if both the query and the materialized view contain aggregates. Here the optimizer determines if the aggregates requested by a query can be derived or computed from one or more aggregates stored in a materialized view. For example, if a query requests AVG(X) and a materialized view contains SUM(X) and COUNT(X), then AVG(X) can be computed as SUM(X) / COUNT(X).

If the grouping compatibility check determined that the rollup of aggregates stored in a materialized view is required, then aggregate computability check determines if it is possible to roll up each aggregate requested by the query using aggregates in the materialized view.

For example, SUM(sales) at the city level can be rolled up to SUM(sales) at the state level by summing all SUM(sales) aggregates in a group with the same state value. However, AVG(sales) cannot be rolled up to a coarser level unless COUNT(sales) is also available in the materialized view. Similarly, VARIANCE(sales) or STDDEV(sales) cannot be rolled up unless COUNT(sales) and SUM(sales) are also available in the materialized view. For example, given the query:

```
SELECT  p.prod_name, AVG(f.dollar_sales) AS avg_sales
FROM    fact f, product p
WHERE   f.prod_key = p.prod_key
GROUP BY p.prod_name;
```

The materialized view *sum_fact_store_prod* can be used to rewrite it provided the join between FACT and STORE is lossless and non-duplicating. Further, the query groups by *prod_name* whereas the materialized view groups by *store_city*, *prod_name*, which means the aggregates stored in the materialized view will have to be rolled up. The optimizer will rewrite the query as:

```
SELECT  mv.prod_name, SUM(mv.sum_sales)/SUM(mv.count_sales) AS avg_sales
FROM    sum_fact_store_prod mv
GROUP BY mv.prod_name;
```

The argument of an aggregate such as SUM can be an arithmetic expression like A+B. The optimizer will try to match an aggregate SUM(A+B) in a query with an aggregate SUM(A+B) or SUM(B+A) stored in a materialized view. In other words, expression equivalence is used when matching the argument of an aggregate in a query with the argument of a similar aggregate in a materialized view. To accomplish this, Oracle converts the aggregate argument expression into a canonical form such that two different but equivalent expressions convert into the same canonical form. For example, A*(B-C), A*B-C*A, (B-C)*A, and -A*C+A*B all convert into the same canonical form and, therefore, they are successfully matched.

When are Constraints and Dimensions Needed?

To clarify when dimensions and constraints are required for the different types of query rewrite, refer to Table 31–2.

Table 31–2 Dimension and Constraint Requirements for Query Rewrite

Rewrite Checks	Dimensions		Primary Key/Foreign Key/Not Null Constraints
Matching SQL Text	Not Required		Not Required
Join Compatibility	Not Required		Required
Data Sufficiency	Required	OR	Required
Grouping Compatibility	Required		Required
Aggregate Computability	Not Required		Not Required

Accuracy of Query Rewrite

Query rewrite offers three levels of rewrite integrity that are controlled by the initialization parameter QUERY_REWRITE_INTEGRITY, which can either be set in your parameter file or controlled using the ALTER SYSTEM or ALTER SESSION command. The three values it can take are:

- ENFORCED

  This is the default mode. The optimizer will only use materialized views which it knows contain fresh data and only use those relationships that are based on enforced constraints.

- TRUSTED

  In TRUSTED mode, the optimizer trusts that the data in the materialized views based on prebuilt tables is correct, and the relationships declared in dimensions and RELY constraints are correct. In this mode, the optimizer uses prebuilt materialized views, and uses relationships that are not enforced as well as those that are enforced. In this mode, the optimizer also ’trusts’ declared but not enforced constraints and data relationships specified using dimensions.

- STALE_TOLERATED

  In STALE_TOLERATED mode, the optimizer uses materialized views that are valid but contain stale data as well as those that contain fresh data. This mode offers the maximum rewrite capability but creates the risk of generating wrong results.

If rewrite integrity is set to the safest level, ENFORCED, the optimizer uses only enforced primary key constraints and referential integrity constraints to ensure that the results of the query are the same as the results when accessing the detail tables directly.

If the rewrite integrity is set to levels other than ENFORCED, then there are several situations where the output with rewrite may be different from that without it.

1. A materialized view can be out of synchronization with the master copy of the data. This generally happens because the materialized view refresh procedure is pending following bulk load or DML operations to one or more detail tables of a materialized view. At some data warehouse sites, this situation is desirable because it is not uncommon for some materialized views to be refreshed at certain time intervals.
2. The relationships implied by the dimension objects are invalid. For example, values at a certain level in a hierarchy do not roll up to exactly one parent value.
3. The values stored in a `PREBUILT` materialized view table may be incorrect.
4. Partition operations such as DROP and MOVE PARTITION on the detail table could affect the results of the materialized view.

Did Query Rewrite Occur?

Since query rewrite occurs transparently, special steps have to be taken to verify that a query has been rewritten. Of course, if the query runs faster, this should indicate that rewrite has occurred but this isn't proof. Therefore, to confirm that query rewrite does occur, use the EXPLAIN PLAN statement.

Explain Plan

The EXPLAIN PLAN facility is used as described in *Oracle8i SQL Reference*. For query rewrite, all you need to check is that the *object_name* column in PLAN_TABLE contains the materialized view name. If it does, then query rewrite will occur when this query is executed.

In this example, the materialized view *store_mv* has been created.

```
CREATE MATERIALIZED VIEW store_mv
 ENABLE QUERY REWRITE
 AS
 SELECT
   s.region, SUM(grocery_sq_ft) AS sum_floor_plan
 FROM store s
 GROUP BY s.region;
```

If EXPLAIN PLAN is used on this SQL statement, the results are placed in the default table PLAN_TABLE.

```
EXPLAIN PLAN
FOR
SELECT  s.region, SUM(grocery_sq_ft)
FROM store s
GROUP BY s.region;
```

For the purposes of query rewrite, the only information of interest from PLAN_TABLE is the OBJECT_NAME, which identifies the objects that will be used to execute this query. Therefore, you would expect to see the object name STORE_MV in the output as illustrated below.

```
SELECT  object_name FROM plan_table;

OBJECT_NAME
------------------------------

STORE_MV
2 rows selected.
```

See Also: For more information about hints, please refer to Chapter , "Using Hints" on page 7-36.

Controlling Query Rewrite

A materialized view is only eligible for query rewrite if the ENABLE QUERY REWRITE clause has been specified, either initially when the materialized view was first created or subsequently via an ALTER MATERIALIZED VIEW command.

The initialization parameters described above can be set using the ALTER SYSTEM SET command. For a given user's session, ALTER SESSION can be used to disable or enable query rewrite for that session only. For example:

```
ALTER SESSION SET QUERY_REWRITE_ENABLED = TRUE;
```

The correctness of query rewrite can be set for a session, thus allowing different users to work at different integrity levels.

```
ALTER SESSION SET QUERY_REWRITE_INTEGRITY = STALE_TOLERATED;
ALTER SESSION SET QUERY_REWRITE_INTEGRITY = TRUSTED;
ALTER SESSION SET QUERY_REWRITE_INTEGRITY = ENFORCED;
```

Rewrite Hints

Hints may be included in SQL statements to control whether query rewrite occurs. Using the NOREWRITE hint with no argument in a query prevents the optimizer from rewriting it.

The REWRITE hint with no argument in a query forces the optimizer to use a materialized view (if any) to rewrite it regardless of the cost.

The REWRITE (*mv1*, *mv2*, ...) hint with argument(s) forces rewrite to select the most suitable materialized view from the list of names specified.

For example, to prevent a rewrite, you can use:

```
SELECT  /*+ NOREWRITE */ s.city, SUM(s.grocery_sq_ft)
FROM store s
GROUP BY s.city;
```

and to force a rewrite using *mv1*, you can use:

```
SELECT /*+ REWRITE (mv1) */ s.city, SUM(s.grocery_sq_ft)
FROM store s
GROUP BY s.city;
```

Guidelines for Using Query Rewrite

The following guidelines will help in getting the maximum benefit from query rewrite. They are not mandatory for using query rewrite and rewrite is not guaranteed if you follow them. They are general rules of thumb.

Constraints

Make sure all inner joins referred to in a materialized view have referential integrity (foreign key - primary key constraints) with additional NOT NULL constraints on the foreign key columns. Since constraints tend to impose a large overhead, you could make them NONVALIDATE and RELY and set the parameter QUERY_REWRITE_INTEGRITY to STALE_TOLERATED or TRUSTED. However, if you set QUERY_REWRITE_INTEGRITY to ENFORCED, all constraints must be enforced to get maximum rewritability.

Dimensions

You can express the hierarchical relationships and functional dependencies in normalized or denormalized dimension tables using the HIERARCHY clause of a dimension. Dimensions can express intra-table relationships which cannot be expressed by any constraints. Set the parameter QUERY_REWRITE_INTEGRITY to TRUSTED or STALE_TOLERATED for query rewrite to take advantage of the relationships declared in dimensions.

Outer Joins

Another way of avoiding constraints is to use outer joins in the materialized view. Query rewrite will be able to derive an inner join in the query, such as (A.a = B.b), from an outer join in the materialized view (A.a = B.b(+)), as long as the rowid of B or column B.b is available in the materialized view. Most of the support for rewrites with outer joins is provided for materialized views with joins only. To exploit it, a materialized view with outer joins should store the rowid or primary key of the inner table of an outer join. For example, the materialized view *join_fact_store_time_oj* stores the primary keys *store_key* and *time_key* of the inner tables of outer joins.

SQL Text Match

If you need to speed up an extremely complex, long-running query, you could create a materialized view with the exact text of the query.

Aggregates

In order to get the maximum benefit from query rewrite, make sure that all aggregates which are needed to compute ones in the targeted query are present in the materialized view. The conditions on aggregates are quite similar to those for incremental refresh. For instance, if AVG(x) is in the query, then you should store COUNT(x) and AVG(x) or store SUM(x) and COUNT(x) in the materialized view. Refer to Table 29–1, "Requirements for Fast Refresh of Materialized Views" on page 29-8.

Grouping Conditions

Aggregating data at lower levels in the hierarchy is better than aggregating at higher levels because lower levels can be used to rewrite more queries. Note, however, that doing so will also take up more space. For example, instead of grouping on state, group on city (unless space constraints prohibit it).

Instead of creating multiple materialized views with overlapping or hierarchically related GROUP BY columns, create a single materialized view with all those GROUP BY columns. For example, instead of using a materialized view that groups by city and another materialized view that groups by month, use a materialized view that groups by city and month.

Use GROUP BY on columns which correspond to levels in a dimension but not on columns that are functionally dependent, because query rewrite will be able to use the functional dependencies automatically based on the DETERMINES clause in a dimension.   For example, instead of grouping on *city_name*, group on *city_id* (as long as there is a dimension which indicates that the attribute *city_id* determines *city_name*, you will enable the rewrite of a query involving *city_name*).

Statistics

Optimization with materialized views is based on cost and the optimizer needs statistics of both the materialized view and the tables in the query to make a cost-based choice. Materialized views should thus have statistics collected using either the ANALYZE TABLE statement or the DBMS_STATISTICS package.

See Also: For more information about collecting statistics, please refer to "Generating Statistics" on page 7-7.

32

Managing Materialized Views

This chapter contains:

Overview of Materialized View Management

The motivation for using materialized views is to improve performance, but the overhead associated with materialized view management can become a significant system management problem. Materialized view management activities include:

- Identifying what materialized views to create initially
- Indexing the materialized views
- Ensuring that all materialized views and materialized view indexes are refreshed properly each time the database is updated
- Checking which materialized views have been used
- Determining how effective each materialized view has been on workload performance
- Measuring the space being used by materialized views
- Determining which new materialized views should be created
- Determining which existing materialized views should be dropped
- Archiving old detail and materialized view data that is no longer useful

This chapter groups these tasks into two areas: warehouse refresh and warehouse advisor, where *warehouse refresh* is concerned with ensuring that the materialized views contain the correct and latest data and the *warehouse advisor* recommends the materialized views to create, retain, and drop.

After the initial effort of creating and populating the data warehouse or data mart, the major administration overhead is the update process, which involves the periodic extraction of incremental changes from the operational systems; transforming the data; verification that the incremental changes are correct, consistent, and complete; bulk-loading the data into the warehouse; and refreshing indexes and materialized views so that they are consistent with the detail data.

The update process must generally be performed within a limited period of time known as the *update window*. The update window depends on the *update frequency* (such as daily or weekly) and is business-dependent. For a daily update frequency, an update window of two to six hours might be typical.

The update window usually displays the time for the following activities:

1. Loading the detail data.
2. Updating or rebuilding the indexes on the detail data.
3. Performing quality assurance tests on the data.
4. Refreshing the materialized views.
5. Updating the indexes on the materialized views.

A popular and efficient way to load data into a warehouse or data mart is to use SQL*Loader with the `DIRECT` or `PARALLEL` option or to use another loader tool that uses the Oracle direct path API.

> **See Also:** See *Oracle8i Utilities* for the restrictions and considerations when using SQL*Loader with the `DIRECT` or `PARALLEL` keywords.

Loading strategies can be classified as *one-phase* or *two-phase*. In one-phase loading, data is loaded directly into the target table, quality assurance tests are performed, and errors are resolved by performing DML operations prior to refreshing materialized views. If a large number of deletions are possible, then storage utilization may be adversely affected, but temporary space requirements and load time are minimized. The DML that may be required after one-phase loading causes multi-table aggregate materialized views to become unusable in the safest rewrite integrity level.

In a two-phase loading process:

- Data is first loaded into a temporary table in the warehouse.
- Quality assurance procedures are applied to the data.
- Referential integrity constraints on the target table are disabled, and the local index in the target partition is marked unusable.
- The data is copied from the temporary area into the appropriate partition of the target table using `INSERT AS SELECT` with the `PARALLEL` or `APPEND` hint.
- The temporary table is dropped.
- The constraints are enabled, usually with the `NOVALIDATE` option.

Immediately after loading the detail data and updating the indexes on the detail data, the database can be opened for operation, if desired. Query rewrite can be disabled by default (with `ALTER SYSTEM SET QUERY_REWRITE_ENABLED = FALSE`) until all the materialized views are refreshed, but enabled at the session level for any users who do not require the materialized views to reflect the data from the latest load (with `ALTER SESSION SET QUERY_REWRITE_ENABLED = TRUE`). However, as long as `QUERY_REWRITE_INTEGRITY = ENFORCED` or `TRUSTED`, this is not necessary as the system ensures that only materialized views with updated data participate in a query rewrite. These packages can be used to refresh any type of materialized view, such as ones containing joins only, or joins and aggregates, or aggregates on single tables.

Warehouse Refresh

When creating a materialized view, you have the option of specifying whether the refresh occurs ON DEMAND or ON COMMIT. To use the fast warehouse refresh facility, the ON DEMAND mode must be specified, then the materialized view can be refreshed by calling one of the procedures in DBMS_MVIEW.

The DBMS_MVIEW package provides three different types of refresh operations.

- `DBMS_MVIEW.REFRESH`

  Refresh one or more materialized views.
- `DBMS_MVIEW.REFRESH_ALL_MVIEWS`

  Refresh all materialized views.
- `DBMS_MVIEW.REFRESH_DEPENDENT`

Refresh all table-based materialized views that depend on a specified detail table or list of detail tables.

See "Manual Refresh Using the DBMS_MVIEW Package" on page 32-6 for more information about this package.

Performing a refresh operation requires temporary space to rebuild the indexes, and can require additional space for performing the refresh operation itself. Fast refresh may also require temporary tables to be created in the user's temporary tablespace.

Some sites may prefer to not refresh all of their materialized views at the same time. Therefore, if you defer refreshing your materialized views, you can temporarily disable query rewrite with `ALTER SYSTEM SET QUERY_REWRITE_ENABLED = FALSE`. Users who still want access to the stale materialized views can override this default with `ALTER SESSION SET QUERY_REWRITE_ENABLED = TRUE`. After refreshing the materialized views, you can re-enable query rewrite as the default for all sessions in the current database instance by setting `ALTER SYSTEM SET QUERY_REWRITE_ENABLED = TRUE`.

Refreshing a materialized view automatically updates all of its indexes; in the case of full refresh, this requires temporary sort space. If insufficient temporary space is available to rebuild the indexes, then you must explicitly drop each index or mark it *unusable* prior to performing the refresh operation.

When a materialized view is refreshed, one of four refresh methods may be specified as shown in the table below.

Refresh Option		Description
COMPLETE	C	Refreshes by recalculating the materialized view's defining query when atomic refresh=TRUE and COMPLETE is the same as FORCE if atomic refresh=FALSE.
FAST	F	Refreshes by incrementally applying changes to the detail tables.
FORCE	?	Uses the default refresh method. If the default refresh method is FORCE, it tries to do a fast refresh. If that is not possible, it does a complete refresh.
ALWAYS	A	Unconditionally does a complete refresh.

Complete Refresh

A complete refresh occurs when the materialized view is initially defined, unless it references a prebuilt table and complete refresh may be requested at any time during the life of the materialized view. Since the refresh involves reading the detail table to compute the results for the materialized view, this can be a very time-consuming process, especially if there are huge amounts of data to be read and processed. Therefore, one should always consider the time required to process a complete refresh before requesting it.

However, there are cases when the only refresh method available is complete refresh because the materialized view does not satisfy the conditions specified in the following section for a fast refresh.

Fast Refresh

Most data warehouses require periodic incremental updates to their detail data. As described in "Schema Design Guidelines for Materialized Views" on page 28-7, you can use the SQL*Loader direct path option, or any bulk load utility that uses Oracle's direct path interface, to perform incremental loads of detail data. Use of Oracle's direct path interface makes fast refresh of your materialized views efficient because, instead of having to recompute the entire materialized view, the changes are added to the existing data. Thus, applying only the changes can result in a very fast refresh time.

The time required to perform incremental refresh is sensitive to several factors:

- Whether the data in the materialized view container table is clustered by a time attribute
- Whether a concatenated index is available on the materialized view keys
- The number of inner joins in the materialized view that have not been declared as part of a referential integrity constraint or `JOIN KEY` declaration in a `CREATE` or `ALTER DIMENSION` statement

The first two factors can be addressed by partitioning the materialized view container by time, like the fact tables, and by creating a local concatenated index on the materialized view keys. The third factor can be addressed by creating dimensions and hierarchies for your schema, and by ensuring that all materialized view inner joins are strict 1:n relationships whenever possible, as described below.

If an incremental load was performed, it is typically much faster to perform a fast refresh than a complete refresh. Warehouse fast refresh is supported in all but the following cases:

- When there is more than one table in an aggregated materialized view, and when any DML on the fact tables, other than a direct load, has occurred since the last full refresh was performed

- When the materialized view contains detail relations that are views or snapshots
- When the materialized view contains `AVG`(x) without `COUNT`(x)
- When the materialized view contains `VARIANCE`(x) without `COUNT`(x) and `SUM`(x)
- When the materialized view contains `STDDEV`(x) without `COUNT`(x) and `SUM`(x)

Note that incremental refresh may perform both inserts and updates to the materialized view. If a new row is inserted, any columns in the materialized view, other than key or measure columns, are set to their default values.

If you want to have a materialized view that can be fast refreshable, even when DML operations such as UPDATE and DELETE are applied to the referenced table, refer to Chapter 29, "Materialized Views", which describes the types of materialized views that allow DML operations, provided a materialized view log exists.

Manual Refresh Using the DBMS_MVIEW Package

Three different refresh procedures are available in the DBMS_MVIEW package for performing ON DEMAND refresh and they each have their own unique set of parameters. To use this package, Oracle8 queues must be available, which means that the following parameters must be set in the initialization parameter file. If queues are unavailable, refresh will fail with an appropriate message.

> **See Also:** See *Oracle8i Supplied Packages Reference* for detailed information about the `DBMS_MVIEW` package. *Oracle8i Replication* explains how to use it in a replication environment.

Required Initialization Parameters for Warehouse Refresh

- JOB_QUEUE_PROCESSES

  The number of background processes. Determines how many materialized views can be refreshed concurrently.

- JOB_QUEUE_INTERVAL

  In seconds, the interval between which the job queue scheduler checks to see if a new job has been submitted to the job queue.

- UTL_FILE_DIR

  Determines the directory where the refresh log is written. If unspecified, no refresh log will be created.

These packages also create a log which, by default, is called refresh.log and is useful in helping to diagnose problems during the refresh process. This log file can be renamed by calling the procedure DBMS_OLAP.SET_LOGFILE_NAME ('log filename').

Refresh Specific Materialized Views

The DBMS_MVIEW.REFRESH procedure is used to refresh one or more materialized views that are explicitly defined in the FROM list. This refresh procedure can also be used to refresh materialized views used by replication, so not all of the parameters are required. The required parameters to use this procedure are:

- The list of materialized views to refresh, delimited by a comma
- The refresh method: A-Always, F-Fast, ?-Force, C-Complete
- Rollback segment to use
- Continue after errors

  When refreshing multiple materialized views, if one of them has an error while being refreshed, the entire job will continue if set to TRUE.
- The following four parameters should be set to FALSE, 0,0,0

  These are the values required by warehouse refresh, since these parameters are used by the replication process.
- Atomic refresh

  If set to TRUE, then warehouse refresh is not used. It uses the snapshot/replication refresh instead. If set to FALSE, the warehouse refresh method is used and each refresh operation is performed within its own transaction.

Therefore, to perform a fast refresh on the materialized view *store_mv*, the package would be called as follows:

```
DBMS_MVIEW.REFRESH('STORE_MV', 'A', '', TRUE, FALSE, 0,0,0, FALSE);
```

Multiple materialized views can be refreshed at the same time and they don't all have to use the same refresh method. To give them different refresh methods, specify multiple method codes in the same order as the list of materialized views (without commas). For example, the following specifies that *store_mv* will be completely refreshed and *product_mv* will receive a fast refresh.

```
DBMS_MVIEW.REFRESH('STORE_MV,PRODUCT_MV', 'AF', '', TRUE, FALSE, 0,0,0, FALSE);
```

Refresh All Materialized Views

An alternative to specifying the materialized views to refresh is to use the procedure DBMS_MVIEW.REFRESH_ALL_MVIEWS. This will result in all materialized views being refreshed. If any of the materialized views fails to refresh, then the number of failures is reported.

The parameters for this procedure are:

- The number of failures
- The datatype number
- The refresh method: A-Always, F-Fast, ?-Force, C-Complete
- Rollback segment to use
- Continue after errors

An example of refreshing all materialized views is:

```
DBMS_MVIEW.REFRESH_ALL_MVIEWS ( failures,'A','',FALSE,FALSE);
```

Refresh Dependent

The third option is the ability to refresh only those materialized views that depend on a specific table using the procedure DBMS_MVIEW. REFRESH_DEPENDENT. For example, suppose the changes have been received for the orders table but not customer payments. The refresh dependent procedure can be called to refresh only those materialized views that reference the ORDER table.

The parameters for this procedure are:

- The number of failures
- The dependent table
- The refresh method: A-Always, F-Fast, ?-Force, C-Complete
- Rollback segment to use
- Continue after errors

  A Boolean parameter. If set to TRUE, the number_of_failures output parameter will be set to the number of refreshes that failed, and a generic error message will indicate that failures occurred. The refresh log will give details of each of the errors, as will the alert log for the instance. If set to FALSE, the default, then refresh, will stop after it encounters the first error, and any remaining materialized views in the list will not be refreshed.

- Atomic refresh

  A Boolean parameter.

In order to perform a full refresh on all materialized views that reference the ORDERS table, use:

```
DBMS_mview.refresh_dependent  (failures, 'ORDERS', 'A', '', FALSE, FALSE );
```

Tips for Refreshing Using Warehouse Refresh

If the process that is executing DBMS_MVIEW.REFRESH is interrupted or the instance is shut down, any refresh jobs that were executing in job queue processes will be requeued and will continue running. To remove these jobs, use the DBMS_JOB.REMOVE procedure.

Materialized Views with Joins and Aggregates

Here are some guidelines for using the refresh mechanism for materialized views with joins and aggregates.

1. The warehouse refresh facility only operates on materialized views containing aggregates.
2. Always load new data using the direct-path option if possible. Avoid deletes and updates because a complete refresh will be necessary. However, you can drop a partition on a materialized view and do a fast refresh.
3. Place fixed key constraints on the fact table, and primary key constraints from the fact table to the dimension table. Doing this enables refresh to identify the fact table, which helps fast refresh.
4. During loading, disable all constraints and re-enable when finished loading.
5. Index the materialized view on the foreign key columns using a concatenated index.
6. To speed up fast refresh, make the number of job queue processes greater than the number of processors.
7. If there are many materialized views to refresh, it is faster to refresh all in a single command than to call them individually.
8. Make use of the "?" refresh method to ensure getting a refreshed materialized view that can be used to query rewrite. If a fast refresh cannot be done, a complete refresh will be performed. Whereas, if a fast refresh had been

requested and there was nothing to do, the materialized view would not be refreshed at all.

9. Try to create materialized views that are fast refreshable because it is quick.
10. If a summary contains data that is based on data which is no longer in the fact table, maintain the materialized view using fast refresh. If no job queues are started, two job queue processes will be started by the refresh. This can be modified by:

    ```
    ALTER SYSTEM SET JOB_QUEUE_PROCESSES = value
    ```

11. In general, the more processors there are, the more job queue processes should be created. Also, if you are doing mostly complete refreshes, reduce the number of job queue processes, since each refresh consumes more system resources than a fast refresh. The number of job queue processes limits the number of materialized views that can be refreshed concurrently. In contrast, if you perform mostly fast refreshes, increase the number of job queue processes.

Refresh of Materialized Views Containing a Single Table with Aggregates

A materialized view which contains aggregates and is based on a single table may be fast refreshable, provided it adheres to the rules in Table 29–1, "Requirements for Fast Refresh of Materialized Views" on page 29-8 when data changes are made using either direct path or SQL DML statements. At refresh time, Oracle detects the type of DML that has been done (direct-load or SQL DML) and uses either the materialized view log or information available from the direct-path to determine the new data. If changes will be made to your data using both methods, then refresh should be performed after each type of data change rather than issuing one refresh at the end. This is because Oracle can perform significant optimizations if it detects that only one type of DML is done. It is therefore recommended that scenario 2 be followed rather than scenario 1.

To improve fast refresh performance, it is highly recommended that indexes be created on the columns which contain the rowids.

Scenario 1

- Direct-load data to detail table
- SQL DML such as INSERT or DELETE to detail table
- Refresh materialized view

Scenario 2

- Direct-load data to detail table
- Refresh materialized view
- SQL DML such as INSERT or DELETE to detail table
- Refresh materialized view

Furthermore, for refresh ON-COMMIT, Oracle keeps track of the type of DML done in the committed transaction. It is thus recommended that the user does not do direct-path load and SQL DML to other tables in the same transaction as Oracle may not be able to optimize the refresh phase.

If the user has a lot of updates to the table, it is better to bunch them in one transaction, so that refresh of the materialized view will be performed just once at commit time rather than after each update. In the warehouse, after a bulk load, the user should enable parallel DML in the session and perform the refresh. Oracle will use parallel DML to do the refresh, which will enhance performance tremendously. There is more to gain if the materialized view is partitioned.

As an example, assume that a materialized view is partitioned and has a parallel clause. The following sequence would be recommended in a data warehouse

1. Bulk load into detail table
2. ALTER SESSION ENABLE PARALLEL DML;
3. Refresh materialized view

Refresh of Materialized Views Containing only Joins

If a materialized view contains joins but no aggregates, then having an index on each of the join column rowids in the detail table will enhance refresh performance greatly since this type of materialized view tends to be much larger than materialized views containing aggregates. For example, referring to the following materialized view:

```
CREATE MATERIALIZED VIEW  detail_fact_mv
       BUILD IMMEDIATE
       REFRESH FASY ON COMMIT
       AS
        SELECT
    f.rowid "fact_rid", t.rowid "time_rid", s.rowid "store_rid",
s.store_key, s.store_name, f.dollar_sales,
f.unit_sales, f.time_key
FROM fact f, time t, store s
```

```
WHERE f.store_key = s.store_key(+) and
f.time_key = t.time_key(+);
```

Indexes should be created on columns FACT_RID, TIME_RID and STORE_RID. Partitioning is highly recommended as is enabling parallel DML in the session before invoking refresh because it will greatly enhance refresh performance.

This type of materialized view can also be fast refreshed if DML is performed on the detail table. It is therefore recommended that the same procedure be applied to this type of materialized view as for a single table aggregate. That is, perform one type of change (direct-path load or DML) and then refresh the materialized view. This is because Oracle can perform significant optimizations if it detects that only one type of change has been done.

Also, it is recommended that the refresh be invoked after each table is loaded, rather than load all the tables and then perform the refresh. Therefore, try to use scenario 2 below for your refresh procedures.

Scenario 1

```
apply changes to fact
apply changes to store
refresh detail_fact_mv
```

Scenario 2

```
apply changes to fact
refresh detail_fact_mv
apply changes to store
refresh detail_fact_mv
```

For refresh ON-COMMIT, Oracle keeps track of the type of DML done in the committed transaction. It is therefore recommended that you do not perform direct-path and conventional DML to other tables in the same transaction because Oracle may not be able to optimize the refresh phase. For example, the following is not recommended:

```
direct path new data into fact
Conventional dml into store
commit
```

One should also try not to mix different types of conventional DML statements if possible. This would again prevent using various optimizations during fast refresh. For example, try to avoid:

```
insert into fact ..
delete from fact ..
commit
```

If many updates are needed, try to group them all into one transaction because refresh will be performed just once at commit time, rather than after each update.

Scenario 1

```
update fact
commit
update fact
commit
update fact
commit
```

Scenario 2

```
update fact
update fact
update fact
commit
```

Note that if, when you use the DBMS_MVIEW package to refresh a number of materialized views containing only joins with the "atomic" parameter set to TRUE, parallel DML is disabled, which could lead to poor refresh performance.

In a data warehousing environment, assuming that the materialized view has a parallel clause, the following sequence of steps is recommended:

1. Bulk load into fact
2. ALTER SESSION ENABLE PARALLEL DML;
3. Refresh materialized view

Recommended Initialization Parameters for Parallelism

The following parameters

- PARALLEL_MAX_SERVERS should be set high enough to take care of parallelism.
- SORT_AREA_SIZE should be less than HASH_AREA_SIZE.
- OPTIMIZER_MODE should equal CHOOSE (cost based optimization).

- OPTIMIZER_PERCENT_PARALLEL should equal 100.

Monitoring a Refresh

While a job is running, a `SELECT * FROM V$SESSION_LONGOPS` statement will tell you the progress of each materialized view being refreshed.

To look at the progress of which jobs are on which queue, use a `SELECT * FROM DBA_JOBS_RUNNING` statement.

The table ALL_MVIEW_ANALYSIS contains the values, as a moving average, for the time most recently refreshed and the average time to refresh using both full and incremental methods.

Refresh will schedule the long running jobs first. Use the refresh log to check what each refresh did.

Tips after Refreshing Materialized Views

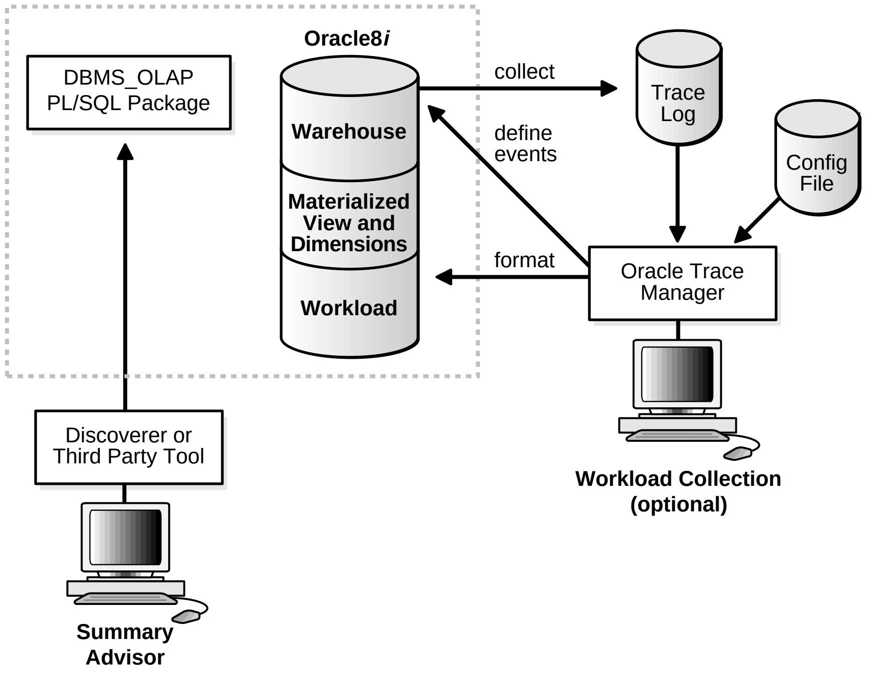
After you have performed a load or incremental load and rebuilt the detail table indexes, you need to re-enable integrity constraints (if any) and refresh the materialized views and materialized view indexes that are derived from that detail data. In a data warehouse environment, referential integrity constraints are normally enabled with the `NOVALIDATE` or `RELY` options. An important decision to make before performing a refresh operation is whether the refresh needs to be recoverable. Because materialized view data is redundant and can always be reconstructed from the detail tables, it may be preferable to disable logging on the materialized view. To disable logging and run incremental refresh non-recoverably, use the `ALTER MATERIALIZED VIEW...NOLOGGING` statement prior to `REFRESH`.

If the materialized view is being refreshed using the ON COMMIT method, then, following refresh operations, the alert log (alert_ <SID>.log) and the trace file (ora_<SID>_number.trc) should be consulted to check that no errors have occurred.

Summary Advisor

To help you select from among the many materialized views that are possible in your schema, Oracle provides a collection of materialized view analysis and advisory functions in the `DBMS_OLAP` package. These functions are callable from any PL/SQL program.

Figure 32–1 Materialized Views and the Advisor



From within the DBMS_OLAP package, several facilities are available to:

- Estimate the size of a materialized view
- Recommend a materialized view
- Recommend materialized views based on collected workload information
- Report actual utilization of materialized views based on collected workload

Whenever the summary advisor is run, with the exception of reporting the size of a materialized view, the results are placed in a table in the database which means that they can be queried, thereby saving the need to keep running the advisor process.

Collecting Structural Statistics

The advisory functions of the `DBMS_OLAP` package require you to gather structural statistics about fact table cardinalities, dimension table cardinalities, and the distinct cardinalities of every dimension LEVEL column, JOIN KEY column, and fact table key column. This can be accomplished by loading your data warehouse, then gathering either exact or estimated statistics with the DBMS_STATS package or the ANALYZE TABLE statement. Because gathering statistics is time-consuming and extreme statistical accuracy is not required, it is generally preferable to estimate statistics. The advisor cannot be used if no dimensions have been defined, which is a good reason for ensuring that some time is spent creating them.

Collection of Dynamic Workload Statistics

Optionally, if you have purchased the *Oracle Enterprise Manager Performance Pack,* then you may also run Oracle Trace to gather dynamic information about your query work load, which can then be used by an advisory function. If Oracle Trace is available, serious consideration should be given to collecting materialized view usage. Not only does it enable the DBA to see which materialized views are in use, but it also means that the advisor may detect some unusual query requests from the users which would result in recommending some different materialized views.

Oracle Trace gathers the following work load statistics for the analysis of materialized views:

- The name of each materialized view selected by query rewrite
- The estimated benefit obtained by using the materialized view, which is roughly the ratio of the fact table cardinality to the materialized view cardinality, adjusted for the need to further aggregate over the materialized view or join it back to other relations
- The "ideal materialized view" that could have been used by the request

Oracle Trace includes two new "point events" for collecting runtime statistics about materialized views: one event that records the selected materialized view names at request execution time, and another event that records the estimated benefit and ideal materialized view at compile time. You can log just these two events for materialized view analysis if desired, or you can join this information with other information collected by Oracle Trace, such as the SQL text or the execution time of the request, if other Trace events are also collected. A collection option in the Oracle Trace Manager GUI provides a way to collect materialized view management statistics.

To collect and analyze the summary event set, you must do the following:

1. Set six initialization parameters to collect data via Oracle Trace. Enabling these parameters incurs some additional overhead at database connection, but is otherwise transparent.

- ORACLE_TRACE_COLLECTION_NAME = oraclesm
- ORACLE_TRACE_COLLECTION_PATH = location of collection files
- ORACLE_TRACE_COLLECTION_SIZE = 0
- ORACLE_TRACE_ENABLE = TRUE turns on Trace collecting
- ORACLE_TRACE_FACILITY_NAME = oraclesm
- ORACLE_TRACE_FACILITY_PATH = location of trace facility files

   For further information on these parameters, refer to the *Oracle Trace Users Guide*.

2. Run the Oracle Trace Manager GUI, specify a collection name, and select the `SUMMARY_EVENT` set. Oracle Trace Manager reads information from the associated configuration file and registers events to be logged with Oracle. While collection is enabled, the workload information defined in the event set gets written to a flat log file.
3. When collection is complete, Oracle Trace automatically formats the Oracle Trace log file into a set of relations, which have the predefined synonyms V_192216243_F_5_E_14_8_1 and V_192216243_F_5_E_15_8_1. The workload tables should be located in the same schema that the subsequent workload analysis will be performed in. Alternatively, the collection file, which usually has an extension of .CDF, can be formatted manually using **otrcfmt**. A manual collection command is illustrated below:

   ```
   otrcfmt    collection_name.cdf    user/password@database
   ```
4. Run the `GATHER_TABLE_STATS` procedure of the `DBMS_STATS` package or `ANALYZE...ESTIMATE STATISTICS` to collect cardinality statistics on all fact tables, dimension tables, and key columns (any column that appears in a dimension `LEVEL` clause or `JOIN KEY` clause of a `CREATE DIMENSION` statement).

Once these four steps have been completed, you will be ready to make recommendations about your materialized views.

Recommending Materialized Views

The analysis and advisory functions for materialized views are `RECOMMEND_MV` and `RECOMMEND_MV_W` in the `DBMS_OLAP` package. These functions automatically recommend which materialized views to create, retain, or drop.

- RECOMMEND_MV uses structural statistics, but not workload statistics, to generate recommendations.
- RECOMMEND_MV_W uses both workload statistics and structural statistics.

You can call these functions to obtain a list of materialized view recommendations that you can select, modify, or reject. Alternatively, you can use the DBMS_OLAP package directly in your PL/SQL programs for the same purpose.

The summary advisor will not be able to recommend summaries if the following conditions are not met:

1. All tables including existing materialized views must have been analyzed as described in step 4 above.
2. Dimensions must exist.
3. The advisor should be able to identify the fact table because it will contain foreign key references to other tables.

> **See Also:** See *Oracle8i Supplied Packages Reference* for detailed information about the DBMS_OLAP package.

Four parameters are required to use these functions:

- Fact table names or null to analyze all fact tables
- The maximum storage that can be used for storing materialized views
- A list or materialized views that you want to retain
- A number between 0 to 100 that specifies the percentage of materialized views that must be retained

A typical call to the package, where the main fact table is called FACT, would be:

```
DBMS_OLAP.RECOMMEND_MV('fact', 100000, '', 10);
```

No workload statistics are used in this example.

The results from calling this package are put in the table MVIEWS$_RECOMMENDATIONS. The contents of this table can be queried or they can be displayed using the SQL file sadvdemo.sql. The output from calling this procedure is the same irrespective of whether the workload statistics are used.

The recommendations can be viewed by calling the procedure DEMO_SUMADV.PRETTYPRINT_RECOMMENDATIONS, but first you need to run sadvdemo.sql. It is suggested that SET SERVEROUTPUT ON SIZE 900000 be used to ensure that all the information can be displayed. A sample recommendation that resulted from calling this package is shown below.

Recommendation Number 1

```
Recommended Action is DROP existing summary GROCERY.QTR_STORE_PROMO_SUM
Storage in bytes is 196020
Percent performance gain is null
Benefit-to-cost ratio is null
```

Recommendation Number 2

```
Recommended Action is RETAIN existing summary GROCERY.STORE_SUM
Storage in bytes is 21
Percent performance gain is null
Benefit-to-cost ratio is null
```

To call the package and use the workload statistics, the only difference is the procedure name that is called. For example, instead of *recommend_mv*, it’s *recommend_mv_w*.

```
DBMS_OLAP.RECOMMEND_MV_W(’fact’, 100000, ’’, 10);
```

Recommendation Number 3

```
Recommendation Number = 3
Recommended Action is CREATE new summary:
SELECT PROMOTION.PROMOTION_KEY, STORE.STORE_KEY, STORE.STORE_NAME,
     STORE.DISTRICT, STORE.REGION , COUNT(*), SUM(FACT.CUSTOMER_COUNT),
     COUNT(FACT.CUSTOMER_COUNT), SUM(FACT.DOLLAR_COST),
COUNT(FACT.DOLLAR_COST),
     SUM(FACT.DOLLAR_SALES), COUNT(FACT.DOLLAR_SALES), MIN(FACT.DOLLAR_SALES),
     MAX(FACT.DOLLAR_SALES), SUM(FACT.RANDOM1), COUNT(FACT.RANDOM1),
     SUM(FACT.RANDOM2), COUNT(FACT.RANDOM2), SUM(FACT.RANDOM3),
     COUNT(FACT.RANDOM3), SUM(FACT.UNIT_SALES), COUNT(FACT.UNIT_SALES)
FROM GROCERY.FACT, GROCERY.PROMOTION, GROCERY.STORE
WHERE FACT.PROMOTION_KEY = PROMOTION.PROMOTION_KEY AND FACT.STORE_KEY =
     STORE.STORE_KEY
GROUP BY PROMOTION.PROMOTION_KEY, STORE.STORE_KEY, STORE.STORE_NAME,
     STORE.DISTRICT, STORE.REGION

Storage in bytes is 257999.999999976
Percent performance gain is .533948057298649
Benefit-to-cost ratio is .00000206956611356085
```

Recommendation Number 4

```
Recommended Action is CREATE new summary:
SELECT STORE.REGION, TIME.QUARTER, TIME.YEAR , COUNT(*),
     SUM(FACT.CUSTOMER_COUNT), COUNT(FACT.CUSTOMER_COUNT),
SUM(FACT.DOLLAR_COST),
     COUNT(FACT.DOLLAR_COST), SUM(FACT.DOLLAR_SALES),
COUNT(FACT.DOLLAR_SALES),
     MIN(FACT.DOLLAR_SALES), MAX(FACT.DOLLAR_SALES), SUM(FACT.RANDOM1),
     COUNT(FACT.RANDOM1), SUM(FACT.RANDOM2), COUNT(FACT.RANDOM2),
     SUM(FACT.RANDOM3), COUNT(FACT.RANDOM3), SUM(FACT.UNIT_SALES),
     COUNT(FACT.UNIT_SALES)
FROM GROCERY.FACT, GROCERY.STORE, GROCERY.TIME
WHERE FACT.STORE_KEY = STORE.STORE_KEY AND FACT.TIME_KEY = TIME.TIME_KEY
GROUP BY STORE.REGION, TIME.QUARTER, TIME.YEAR

Storage in bytes is 86
Percent performance gain is .523360688578368
Benefit-to-cost ratio is .00608558940207405
```

Estimating Materialized View Size

Since a materialized view occupies storage space in the database, it is helpful to know how much space will be required before it is created. Rather than guess or wait until it has been created and then discoverer that insufficient space is available in the tablespace, use the package DBMS_ESTIMATE_SIZE. Calling this procedure instantly returns an estimate of the size in bytes that the materialized view is likely to occupy.

The parameters to this procedure are:

- the name for sizing
- the SELECT statement

and the package returns:

- the number of rows it expects in the materialized view
- the size of the materialized view in bytes

In the example shown below, the query that will be specified in the materialized view is passed into the ESTIMATE_SUMMARY_SIZE package. Note that the SQL statement is passed in without a ";".

```
DBMS_OLAP.estimate_summary_size ('simple_store',
   'SELECT
    product_key1, product_key2,
    SUM(dollar_sales) AS sum_dollar_sales,
    SUM(unit_sales) AS sum_unit_sales,
    SUM(dollar_cost) AS sum_dollar_cost,
    SUM(customer_count) AS no_of_customers
    FROM fact GROUP BY product_key1, product_key2' ,
       no_of_rows, mv_size  );
```

The procedure returns two values, an estimate for the number of rows and the size of the materialized view in bytes, as shown below.

```
No of Rows: 17284
Size of Materialized view (bytes): 2281488
```

Is a Materialized View Being Used?

One of the major administrative problems with materialized views is knowing whether they are being used. Materialized views could be in regular use or they could have been created for a one-time problem that has now been resolved. However, the usergroup who requested this level of analysis might never have told the DBA that it was no longer required, so the materialized view remains in the database occupying storage space and possibly being regularly refreshed.

If the Oracle Trace option is available, then it can advise the DBA which materialized views are in use, using exactly the same procedure as for collecting workload statistics. Trace collection is enabled and in this case the collection period is likely to be longer that for query collection because Trace will only report on materialized views that were used while it was collecting statistics. Therefore, if too small a window is chosen, not all the materialized views that are in use will be reported.

Once you are satisfied that you have collected sufficient data, the data is formatted by Oracle Trace, just as if it were workload information, and then the package EVALUATE_UTILIZATION_W is called. It analyses the data and then the results are placed in the table MVIEWS$_EVALUATIONS.

In the example below, the utilization of materialized views is analyzed and the results are displayed.

```
DBMS_OLAP.EVALUATE_UTILIZATION_W();
```

Note that no parameters are passed into the package.

Shown below is a sample output obtained by querying the table MVIEW$EVALUATIONS which is providing the following information:

- Materialized view owner and name
- Rank of this materialized view in descending benefit-to-cost ratio
- Size of the materialized view in bytes
- The number of times the materialized view appears in the workload
- The cumulative benefit is calculated each time the materialized view is used as
- The benefit-to-cost ratio is calculated as the incremental improvement in performance to the size of the materialized view

```
MVIEW_OWNER MVIEW_NAME          RANK    SIZE FREQ CUMULATIVE    BENEFIT
----------- ------------------- ----- ------ ---- ---------- ----------
GROCERY     STORE_MIN_SUM       1        340    1       9001 26.4735294
GROCERY     STORE_MAX_SUM       2        380    1       9001 23.6868421
GROCERY     STORE_STDCNT_SUM    3       3120    1 3000.38333 .961661325
GROCERY     QTR_STORE_PROMO_SUM 4     196020    2          0          0
GROCERY     STORE_SALES_SUM     5        340    1          0          0
GROCERY     STORE_SUM           6         21   10          0          0
```

Index