Oracle8™

Tuning

Release 8.0

December, 1997

Part No. A58246-01

ORACLE®

Enabling the Information Age™

Oracle8™ Tuning

Part No. A58246-01

Release 8.0

Primary Author: Rita Moran

Primary Contributors: Graham Wood, Anjo Kolk, Gary Hallmark

Contributors: Tomohiro Akiba, David Austin, Andre Bakker, Allen Brumm, Dave Colello, Carol Colrain, Benoit Dageville, Dean Daniels, Dinesh Das, Michael Depledge, Joyce Fee, John Frazzini, Jyotin Gautam, Jackie Gosselin, Scott Gossett, John Graham, Todd Guay, Mike Hartstein, Scott Heisey, Alex Ho, Andrew Holdsworth, Hakan Jakobssen, Sue Jang, Robert Jenkins, Jan Klokkers, Paul Lane, Dan Leary, Tirthankar Lahiri, Juan Loaiza, Diana Lorentz, George Lumpkin, Roderick Manalac, Sheryl Maring, Ravi Mirchandaney, Ken Morse, Jeff Needham, Kotaro Ono, Cetin Ozbutun, Orla Parkinson, Doug Rady, Mary Rhodes, Ray Roccaforte, Hari Sankar, Leng Leng Tan, Lawrence To, Dan Tow, Peter Vasterd, Sandy Venning, Radek Vingralek, Bill Waddington, Mohamed Zait

Graphic Designer: Valarie Moore

Contents

6 Designing Data Warehouse Applications

Part III Optimizing Database Operations

7 Tuning Database Operations

8 Optimization Modes and Hints

9 Tuning Distributed Queries

10 Data Access Methods

11 Oracle8 Transaction Modes

15 Tuning I/O

16 Tuning Networks

17 Tuning the Operating System

18 Tuning Resource Contention

20 Understanding Parallel Execution Performance Issues

21 Diagnosing Parallel Execution Performance Problems

25 Using Oracle Trace

26 Registering Applications

Send Us Your Comments

Oracle8™ Tuning, Release 8.0

Part No. A58246-01

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

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

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

- infodev@us.oracle.com
- FAX - 650-506-7228. Attn: Oracle8 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 itself, 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: *Oracle8 Tuning* contains information that describes the features and functionality of the Oracle8 and the Oracle8 Enterprise Edition products. Oracle8 and Oracle8 Enterprise Edition have the same basic features. However, several advanced features are available only with the Enterprise Edition, and some of these are optional. For example, to use application failover, you must have the Enterprise Edition and the Parallel Server Option.

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

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, the operating system, and application design before reading this manual.

Structure

This manual contains six parts:

Part 1: Introduction

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 Method	This chapter presents the recommended tuning method, and outlines its steps in order of priority.
Chapter 3: Diagnosing Performance Problems in an Existing System	This chapter provides an overview of performance factors in existing systems that have been properly designed.
Chapter 4: Overview of Diagnostic Tools	This chapter introduces the full range of diagnostic tools available for monitoring production systems and determining performance problems.

Part 2: Designing and Developing for Performance

Chapter 5: Evaluating Your System's 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 6: Designing Data Warehouse Applications	This chapter introduces integrated Oracle8 features for tuning enterprise-scale data warehouses.

Part 3: Writing Efficient SQL Statements

Chapter 7: Tuning Database Operations	This chapter explains the fundamentals of tuning database operations.
Chapter 8: Optimization Modes and Hints	This chapter explains when to use the available optimization modes and how to use hints to enhance Oracle performance.
Chapter 9: Tuning Distributed Queries	This chapter provides guidelines for tuning distributed queries.
Chapter 10: Data Access Methods	This chapter provides an overview of data access methods that can enhance performance, and warns of situations to avoid.
Chapter 11: Oracle8 Transaction Modes	This chapter describes the different methods in which read consistency is performed.
Chapter 12: Managing SQL and Shared PL/SQL Areas	This chapter explains the use of shared SQL to improve performance.

Part 4: Optimizing Oracle Instance Performance

Chapter 13: Tuning CPU Resources	This chapter describes how to identify and solve problems with CPU resources.
Chapter 14: Tuning Memory Allocation	This chapter explains how to allocate memory to database structures. Proper sizing of these structures can greatly improve database performance.
Chapter 15: Tuning I/O	This chapter explains how to avoid I/O bottlenecks that could prevent Oracle from performing at its maximum potential.
Chapter 16: 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 17: Tuning the Operating System	This chapter explains how to tune the operating system for optimal performance of Oracle.
Chapter 18: Tuning Resource Contention	This chapter explains how to detect and reduce contention that affects performance.

Part 5: Optimizing Parallel Execution

Chapter 19: Tuning Parallel Execution	This chapter explains how to use parallel execution features for improved performance.
Chapter 20: Understanding Parallel Execution Performance Issues	This chapter provides a conceptual explanation of parallel execution performance issues.
Chapter 21: Diagnosing Parallel Execution Performance Problems	This chapter explains how to diagnose and solve performance problems in parallel execution.

Part 6: Performance Diagnostic Tools

Chapter 22: The Dynamic Performance Views	This chapter describes views that are of the greatest use for both performance tuning and ad hoc investigation
Chapter 23: The EXPLAIN PLAN Command	This chapter shows how to use the SQL command EXPLAIN PLAN, and format its output.
Chapter 24: 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 25: Using Oracle Trace	This chapter provides an overview of Oracle Trace usage and describes the Oracle Trace initialization parameters.
Chapter 26: Registering Applications	This chapter describes how to register an application with the database and retrieve statistics on each registered module or code segment.

Related Documents

This manual assumes you have already read *Oracle8 Concepts*, the *Oracle8 Application Developer's Guide*, and *Oracle8 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 commands, 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 *d*	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.

Your Comments Are Welcome

We value and appreciate your comments as an Oracle user and reader of the manuals. As we write, revise, and evaluate our documentation, your opinions are the most important input we receive. Please use the reader's comment form to tell us what you like and dislike about this manual or other Oracle manuals. If the form is not available, please use the following address:

- infodev@us.oracle.com
- FAX - 650-506-7228. Attn: Oracle8 Tuning
- postal service:

  Oracle Corporation
  Server Technologies Documentation
  500 Oracle Parkway, 4OP12
  Redwood Shores, CA 94065
  U.S.A.

Part I

Introduction

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

- Chapter 1, “Introduction to Oracle Performance Tuning”
- Chapter 2, “Performance Tuning Method”
- Chapter 3, “Diagnosing Performance Problems in an Existing System”
- Chapter 4, “Overview of Diagnostic Tools”

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.

This chapter gives an overview of tuning issues. Topics in this chapter include:

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

What Is Performance Tuning?

Performance must be built in! Performance tuning cannot be performed optimally after a system is put into production. To achieve performance targets of response time, throughput, and constraints you must tune application analysis, design, and implementation. This section introduces some fundamental concepts:

- 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 various different tables. Throughput becomes more of an issue when an application must support a large number of users running DSS queries.

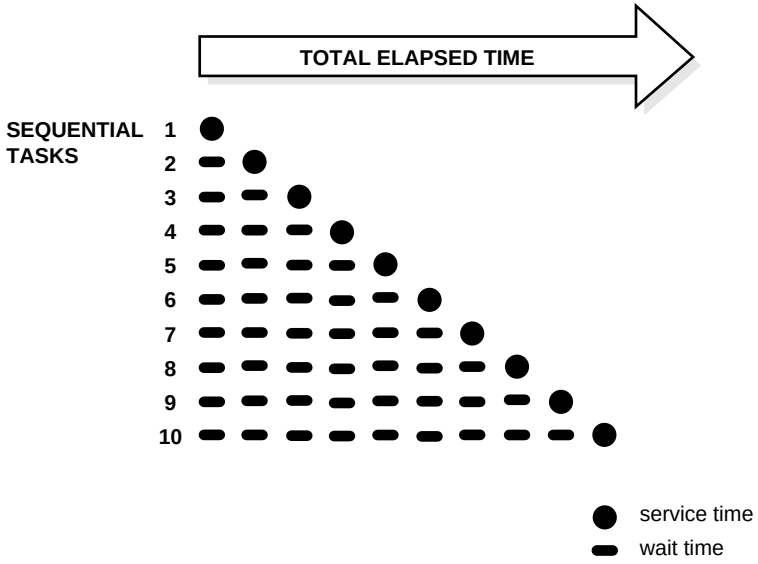
Response Time

Because response time equals service time plus wait time, you can increase performance in two ways:

- by reducing service time
- 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 will 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

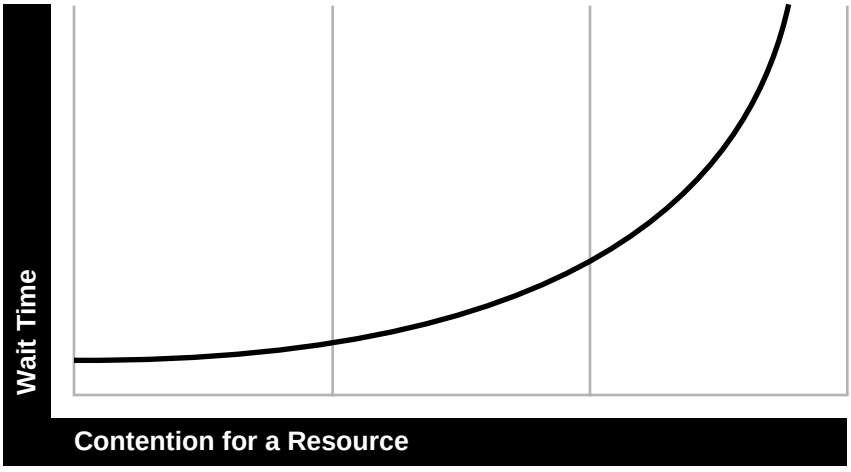
System throughput equals the amount of work accomplished in a given amount of time. Two techniques of increasing throughput exist:

- 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 will go up 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



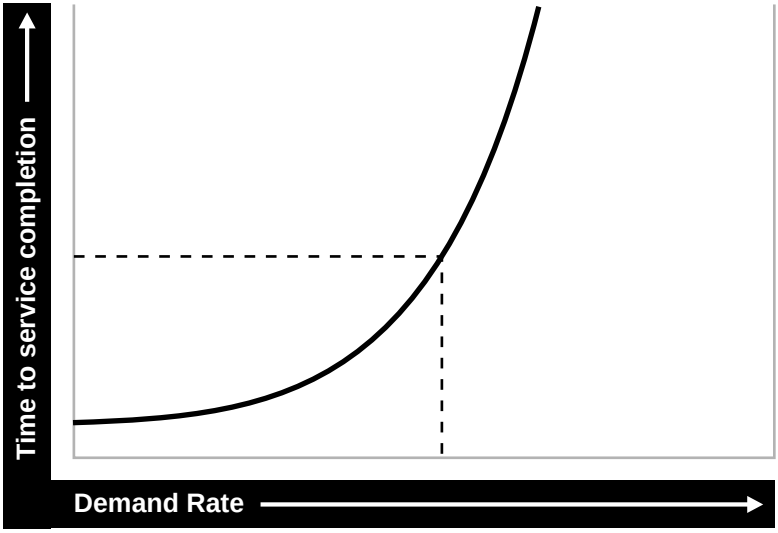
Critical Resources

Resources such as CPUs, memory, I/O capacity, and network bandwidth are key to reducing service time. Added resources make possible higher throughput and 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.

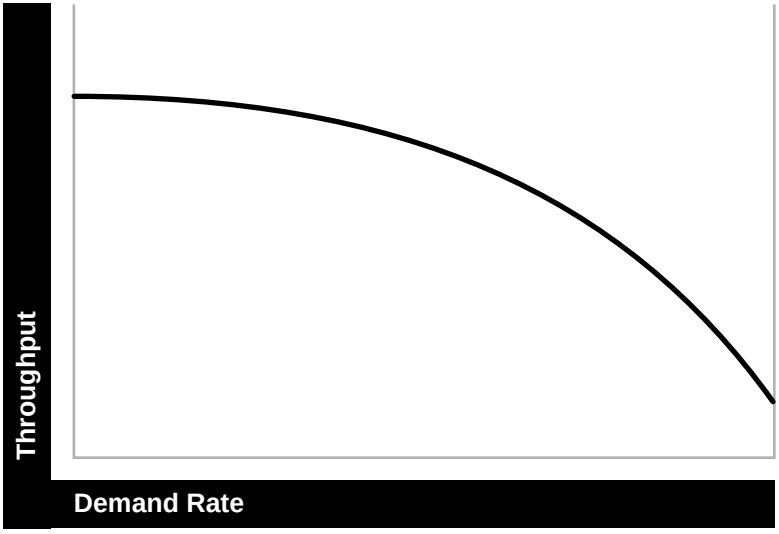
Effects of Excessive Demand

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



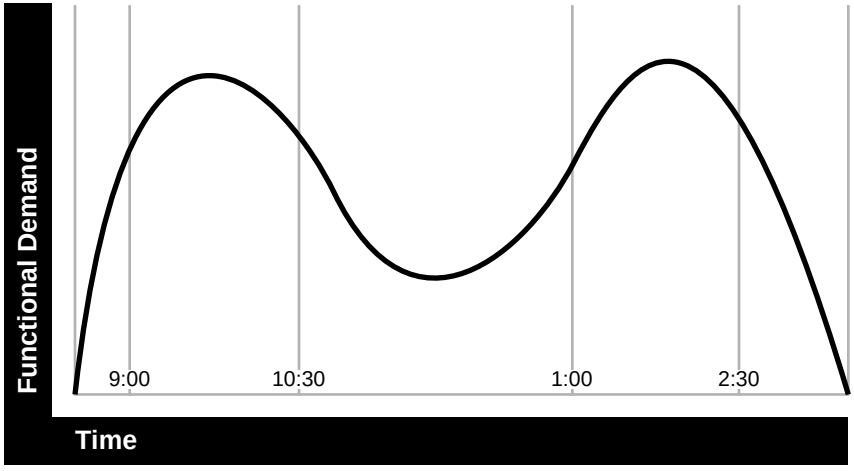
Adjustments to Relieve Problems

Performance problems can be relieved by making the following adjustments:

adjusting unit consumption	You can relieve some problems by using less resource 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 abated 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, you will have multiple resources you can use.

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

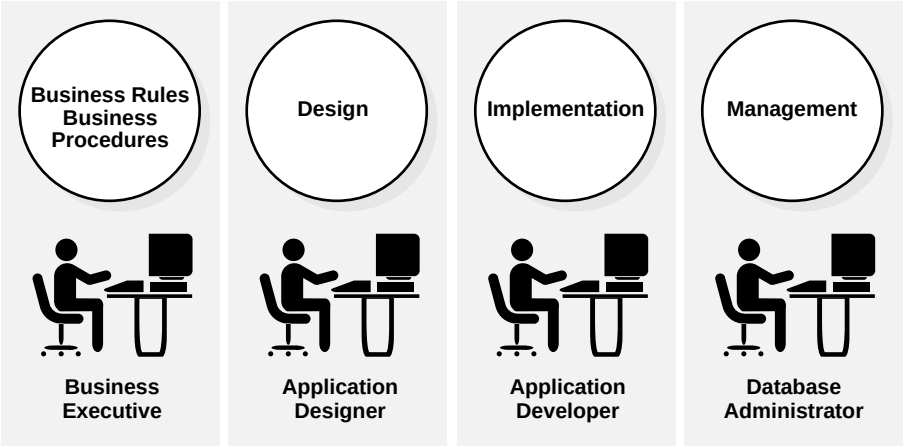
Figure 1–5 Adjusting Capacity and Functional Demand



Who Tunes?

Everyone involved with the system has some role in the tuning process. 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 kinds of rules and procedures that can influence the performance of the whole system.
- Application designers must design around potential performance bottlenecks. They must communicate the system design so that everyone can understand the flow of data in an application.
- Application developers must communicate the implementation strategies they choose so that modules and SQL statements can be quickly and easily identified during statement tuning.
- Database administrators (DBAs) must carefully monitor and document system activity so that they can identify and correct unusual system performance.
- Hardware and software administrators must document and communicate the configuration of the system so that everyone can design and administer the system effectively.

Decisions made in application development and design have the most impact on performance. Once the application is deployed, the database administrator usually has the primary responsibility for tuning—within the limitations of the design.

See Also: Chapter 3, "Diagnosing Performance Problems in an Existing System" for keys that can help database administrators (DBAs) to identify performance problems and solve them reactively.

Setting Performance Targets

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

When designing your system, set a specific goal: for example, an order entry response time of less than three seconds. If the application does not meet that goal, identify the bottleneck causing the slowdown (for example, I/O contention), determine the cause, and take corrective action. During development, you should 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 determined the bottlenecks, you may have to sacrifice some other areas 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, if you have clearly defined goals for performance, the decision on what to trade for higher performance is simpler because you have identified the most important areas.

Setting User Expectations

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

If a DBA should promise 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 will 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 or not the criteria are being met, and your ability to take corrective action to overcome any 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 who are responsible for solving performance problems must keep a wide view of the all the factors that together determine response time. The perceived area of performance problems is frequently not the actual source of the problem. 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 find 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. Keeping a history of the application's performance over time enables you to make useful comparisons. With data about actual resource consumption for a range of loads, you can conduct objective scalability studies and from these predict the resource requirements for load volumes you may anticipate in the future.

See Also: Chapter 4, "Overview of Diagnostic Tools"

2

Performance Tuning Method

Methodology is key to success in performance tuning. Different tuning strategies offer diminishing returns, and it is important to use the strategies with the maximum gains first. Furthermore, systems with different purposes, such as online transaction processing systems and decision support systems, may require different approaches.

Topics in this chapter include

- When Is Tuning Most Effective?
- Prioritized Steps of the Tuning Method
- How to Apply the Tuning Method

See Also: "Oracle Expert" on page 4-12. Oracle Expert automates the process of collecting and analyzing data, and contains rules that provide database tuning recommendations, implementation scripts, and reports.

When Is Tuning Most Effective?

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

- Proactive Tuning While Designing and Developing a System
- Reactive Tuning to Improve a Production System

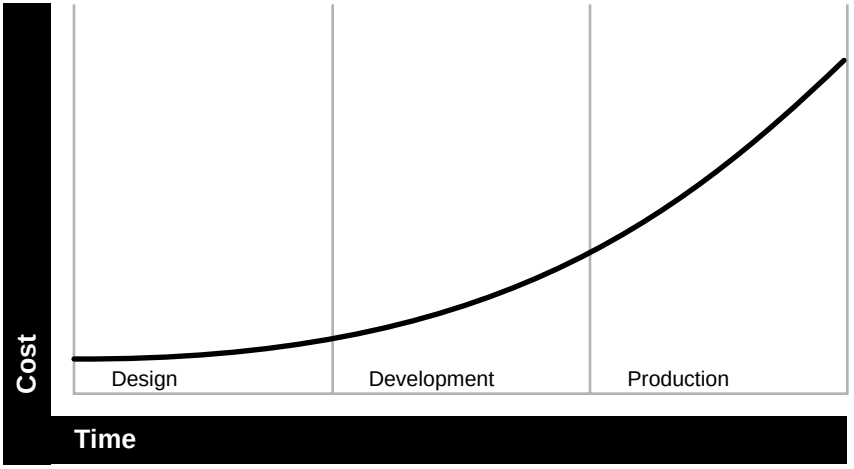
Proactive Tuning While Designing and Developing a System

By far the most effective approach to tuning is to work proactively. Start off at the beginning of the method described in this chapter, and work your way down.

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

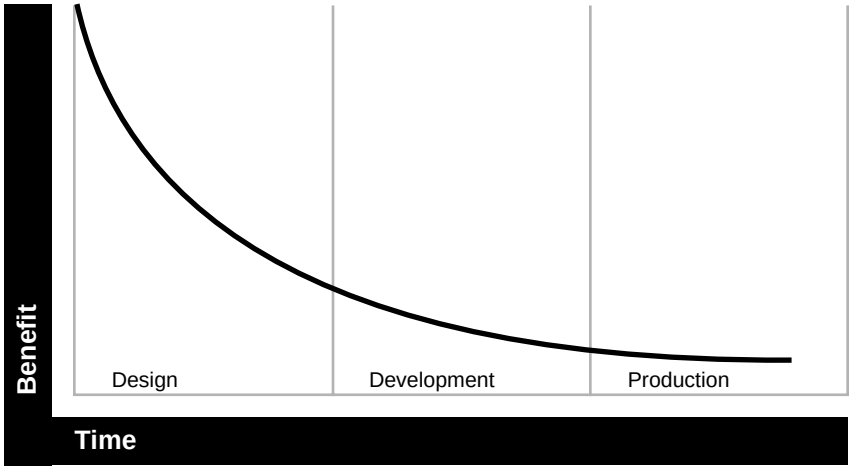
By designing a system to perform well, you can minimize its eventual cost and frustration. Figure 2–1 illustrates the relative *cost* of tuning during the life of an application.

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



As you can see, the most effective time to tune is during the design phase: you get the maximum benefit at the lowest cost.

Reactive Tuning to Improve a Production System

Many people believe the tuning process begins when users complain about poor response time. This is usually too late in the process 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.

Consider, for example, a bank which employs one teller and one manager. It has a business rule that the manager must approve any withdrawals exceeding $20. Upon investigation, you may find that there is a long queue of customers, and deduce that 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. Regardless of how carefully you may tune the system using the existing business rule, getting better performance will be very expensive.

Upon stepping back, you may see that a change to the business rule may be necessary to make the system more scalable. If you change the rule such that the manager need only approve withdrawals exceeding $150, you have come up with 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 nonetheless possible to work reactively to 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 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.

Note that 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, which contains chapters that describe in detail how to tune CPU, memory, I/O, networks, contention, and the operating system.

Oracle8 Concepts: To find performance bottlenecks quickly and easily and determine the corrective action for a production system, you must have a firm understanding of Oracle Server architecture and features.

Prioritized Steps of the Tuning Method

The recommended method for tuning an Oracle database prioritizes steps in order of diminishing returns: steps with the greatest impact on performance are listed first. For optimal results, therefore, tackle 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

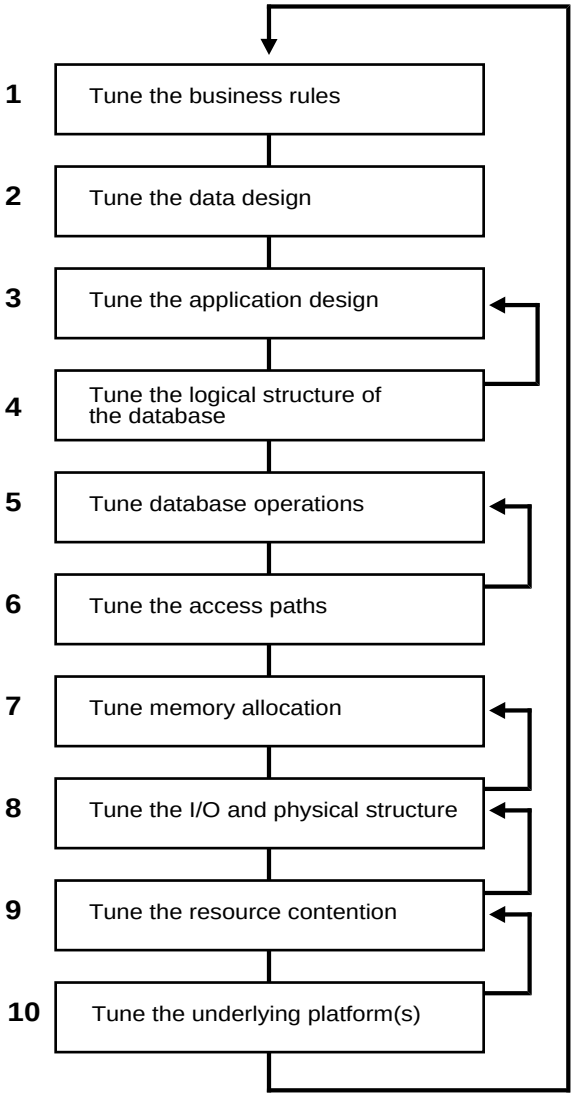
Step 10: Tune the Underlying Platform(s)

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

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

Figure 2–3 illustrates the tuning method:

Figure 2–3 The Tuning Method



Decisions you make in one step may influence subsequent steps. For example, in step 5 you may rewrite some of your SQL statements. These SQL statements may have significant bearing on parsing and caching issues addressed in step 7. Also,

disk I/O, which is tuned in step 8, depends on the size of the buffer cache, which is tuned in step 7. Although the figure shows a loop back to step 1, you may need to loop back 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 or not to use a multithreaded 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. People commonly get in too deep when they write the business functions of an application. They document an implementation, rather than simply the function that must be performed. If business executives use care in abstracting the business function or requirement from the implementation, then designers have a wider field from which to choose the 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 up pieces of paper. Whereas it would be very difficult to print up a million checks per day, it would be relatively easy to record that many direct deposit payments on a tape which 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, making use of a multithreaded 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, in which data is analyzed to ensure that no redundant data will be held anywhere. One fact should be stated in one and only one place in the database. Once the data is carefully normalized, however, you may need to denormalize it for performance reasons. You might, for example, decide that the database should hold frequently required summary values. 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 include the total value of 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 the avoidance of contention on data. Consider a database 1 terabyte in size, on which a 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: localize it to each process, to each instance, and to each partition. Contention begins when access becomes remote, and the amount of contention determines scalability.

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.

Consider two Oracle8 enhancements that can help you to tune the data design to avoid contention:

- Consider whether or not to partition your data.
- Consider whether to use local or global indexes.

See Also: Chapter 2, “Performance Tuning Method”
"Partitioning Data" on page 19-23
Oracle8 Concepts for discussions of partitioning and indexes

Step 3: Tune the Application Design

Business executives and application designers need to 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 course of the day.

At this level also, you can consider configuration of individual processes. For example, some PC users may be accessing the central system using mobile agents, whereas other users may be directly connected. Although they are running on the same system, the architecture 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 designing the use and location of indexes, the sequence generator, and clusters.

See Also: "Using Indexes" on page 10-2

Step 5: Tune Database Operations

System designers and application developers must understand Oracle's query processing mechanism to write effective SQL statements. Chapter 8, "Optimization Modes and Hints", discusses Oracle's query optimizer and how to write statements that achieve the fastest results.

Before tuning the Oracle Server itself, be certain that your application is taking full advantage of the SQL language and the Oracle features designed to speed 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

See Also: "Part 3: Optimizing Database Operations"

Step 6: Tune the Access Paths

Ensure that there is efficient access to data. 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 mean revisiting your design after you have built the database. You may want to do more normalization or create alternative indexes at this point. Upon testing the application you may find that you're still not obtaining the required response time. Look for more ways to improve the design.

See Also: Chapter 10, "Data Access Methods"

Step 7: Tune Memory Allocation

Appropriate allocation of memory resources to Oracle memory structures can have a large impact on performance.

Oracle8 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 multithreaded server)

You can explicitly set memory allocation for the following structures:

- Buffer cache
- Log buffer
- Sequence caches

Proper allocation of memory resources can improve cache performance, reduce parsing of SQL statements, and reduce paging and swapping.

Process local areas include:

- Context areas (for systems not running a multithreaded 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 14, "Tuning Memory Allocation"
Oracle8 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. Oracle Server, however, is designed so that its performance need not be unduly limited by I/O. Tuning I/O and physical structure involves these procedures:

- Distributing data so that I/O is distributed, thus avoiding disk contention
- Storing data in data blocks for best access: setting the right number of free lists, and proper values for PCTFREE and PCTUSED
- Creating extents large enough for your data so as to avoid dynamic extension of tables, which would hurt high-volume OLTP applications
- Evaluating the use of raw devices

See Also: Chapter 15, "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 kinds of contention:

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

See Also: Chapter 18, "Tuning Resource Contention"

Step 10: Tune the Underlying Platform(s)

See your platform-specific Oracle documentation to investigate 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 17, "Tuning the Operating System"

How to Apply 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
- Avoid Common Errors
- Stop Tuning When the Objectives Are Met
- Demonstrate Meeting the Objectives

Set Clear Goals for Tuning

Never begin tuning without having first established clear objectives: you cannot succeed if there is no definition of "success."

"Just make it go as fast as you can" may sound like an objective, but it will be very difficult to determine whether this has been achieved. It will be even more difficult to tell whether your results have met the underlying business requirements. A more useful statement of objectives is the following: "We need to have as many as 20 operators each entering 20 orders per hour, and the packing lists 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 bear in mind 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 reproducible cases. For example, if you manage to identify a single SQL statement that is causing a performance problem, then run both the original and the revised version of that statement in SQL*Plus (with the SQL trace facility or Oracle Trace enabled) so that you can see statistically the difference in performance. In many cases, a tuning effort can succeed simply by identifying one SQL statement that was causing the performance problem.

If you must cut a 4-hour run down to 2 hours duration, you will probably find that repeated timings take too long. Perform your initial trials against a test environment that exhibits a profile similar to the real one. For example, you could impose some additional restrictive condition such as processing one department instead of all 500 of them. The ideal test case will run for more than 1 minute, so that improvements can be seen intuitively, as well as measured using timing features. It should run for less than 5 minutes, however, so that test execution does not consume an excessive proportion of the time available.

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.

Bear in mind that Oracle's caching algorithms mean that the first time data is visited there is an additional overhead. Thus, if two approaches are tried one after the other, the second will always have a tactical advantage: data which it would otherwise have had to read from disk may be left in the cache.

Keep Records

Keep records of the effect of each change: incorporate record keeping in the script being used to run the test. You should automate the testing process for a number of reasons:

- For cost effectiveness in terms of the tuner's ability to conduct tests quickly
- To ensure that the tests are conducted in the same way, using the same instrumentation, for each hypothesis being tested

Anecdotal results from tests should be checked against the objective data before being accepted.

Avoid Common Errors

A common error made by inexperienced tuners is to cling to preconceived notions about what may be causing the problem at hand. The next most common error is to try various approaches at random.

Each time you think that you are onto something, try explaining it to someone else. Often you yourself will spot mistakes, simply from having gone through the discipline of articulating your ideas. For best results you should build a team of people to resolve performance problems. While a performance tuner can tune SQL statements without knowing the application in detail, the team should include someone who does understand the application and who can validate the solutions that the SQL tuner may devise.

Avoid Rash Actions

Beware of doing something rash. Once you have a hypothesis, you may be tempted to implement it globally throughout the system and then wait to see the results. You can ruin a perfectly good system in this way!

Avoid Preconceptions

Try to avoid preconceptions when you come to a tuning problem. Get users to tell you the symptoms they perceive—but do not expect them to know why the problem exists.

One user, for example, had serious system memory problems over a long period of time. In the morning the system ran well, but performance then dropped off very rapidly. A consultant called in to tune the system was told that a PL/SQL memory leak was the cause. As it turned out, this was not at all the problem. Rather, the user had set SORT_AREA_SIZE to 10 MB on a machine with 64 MB of memory, and had 20 users. When users came on to the system, the first time they did a sort they would get their sort area. The system thus was burdened with 200 MB of virtual memory and was hopelessly swapping and paging.

Many people will speculate about the cause of the problem. Ask questions of those affected, and of those responsible for the system. Listen to the symptoms that users describe, but do not accept prima facie their notions as to the cause!

Avoid “Hit and Hope”

Avoid seizing on panaceas that may be bandied about in database folklore, such as “set the GO_FASTER parameter and everything will work faster.” Be wary of apocryphal tales—such as the false notion that all tables must be in a single extent for performance to be acceptable.

Stop Tuning When the 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 the 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

3

Diagnosing Performance Problems in an Existing System

This chapter provides an overview of factors affecting performance in existing systems that have been properly designed. Note, however, that tuning these factors cannot compensate for poor design!

- Tuning Factors for a Well-Designed Existing System
- Insufficient CPU
- Insufficient Memory
- Insufficient I/O
- Network Constraints
- Software Constraints

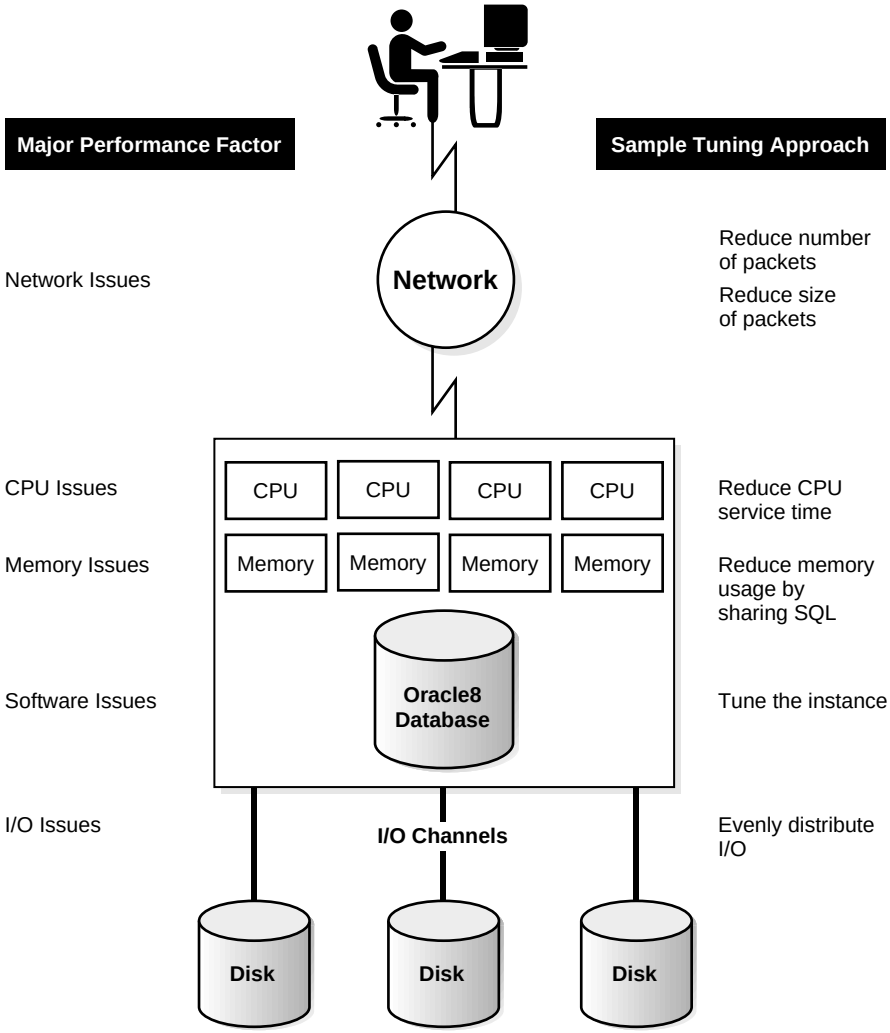
Later chapters discuss each of these factors in depth.

Tuning Factors for a Well-Designed Existing System

The following figure illustrates the many factors involved in Oracle system performance, when considering a well-designed existing application.

Attention: Tuning these factors is effective only *after* you have tuned the business process and the application, as described in "Performance Tuning Method". Tuning these factors cannot make up for the performance gains that must be designed into the system.

Figure 3–1 Major Performance Factors in a Well-designed System



Performance problems tend to be interconnected rather than isolated. The following table provides a key to performance factors in an existing system, and the areas in which symptoms may appear. Buffer cache problems, for example, may show up as issues of CPU, memory, or I/O. Tuning the buffer cache for CPU may in turn improve I/O.

Table 3–1 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, the CPU resource is completely used up, service time is too high and you want to achieve more. Alternatively, you have too much idle time, want to achieve more, and the CPU is not completely used up. There is room to do more: you need to determine why so much time is spent waiting.

To diagnose insufficient CPU, you must check CPU utilization by your entire system, not only utilization by Oracle Server processes. At the beginning of a workday, for example, the mail system may consume a very large amount of the available CPU while employees check their messages. Later in the day, the mail system may be much less heavily used, and its CPU utilization will drop accordingly.

Workload is a very 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 understandable and 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 not room for peak workload. You have a CPU problem if idle time and time waiting for I/O are both close to zero (less than 5%) at a normal or low workload.

See Also: Chapter 13, "Tuning CPU Resources"

Insufficient Memory

Sometimes a memory problem may be detected as an I/O problem. There are two kinds of memory requirements: Oracle and system. Oracle memory requirements affect the system requirements. Memory problems may be the cause of all the paging and swapping that goes on in the machine. Make sure that your system does not start swapping and paging. The whole 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 internal memory. To achieve this, you can trim some elements of the Oracle requirements, most likely buffer cache, shared pool, or redo log buffer. On the system level you can trim 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 usage might be by sharing SQL.

See Also: Chapter 14, "Tuning Memory Allocation"

Insufficient I/O

Be sure to distribute I/O evenly across disks and channels. I/O resources involve:

- Channel bandwidth: number of I/O channels
- Device bandwidth: number of disks
- Device latency: latency will be part of your wait time

I/O problems may result from limitations of your hardware configuration. Your system needs enough disks and SCSI busses to support the transaction throughput you desire. You can evaluate the configuration by figuring the number 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 too high, the most common problem is that wait time has gone up (response time = service time + wait time). If wait time goes up, it means that there are too many I/O requests for this device. If service time goes up, this normally means that the I/O requests are larger, so you write more bytes.

The different background processes (DBWR, ARCH, and so on) perform different kinds of I/O, and each process has different I/O characteristics. Some 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 over different devices.

Mirroring can also be a cause of I/O bottlenecks unless the data is mirrored on the same number of disks as the database itself.

See Also: Chapter 15, “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 lots of small packets.
- Transmission time

As the number of users and the demand rises, the network can sometimes quietly become the bottleneck in an application. You may be spending a lot of time waiting for network availability. Use available operating system tools to see how busy your network is.

See Also: Chapter 16, "Tuning Networks"

Software Constraints

Operating system software determines:

- The maximum number of processes you can support
- The maximum number of processes you can connect

Before you can tune Oracle effectively, you must ensure that the operating system is at its peak performance. 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 17, "Tuning the Operating System".

4

Overview of Diagnostic Tools

This chapter introduces the full range of diagnostic tools that are available 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
- The SQL Trace Facility and TKPROF
- Supported Scripts
- Application Registration
- Oracle Enterprise Manager 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. Note that many of these sources may be transient. They include:

- Data Volumes
- Online Data Dictionary
- Operating System Tools
- Dynamic Performance Tables
- 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 that can tell you how many transactions were performed, 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 (the throughput). Where such rows contain a time stamp, you can query the table and use a graphics package to plot the throughput against date and time. Such a date-time stamp need not be apparent to the rest of the application.

If your application does *not* contain an audit table, you might not want to add one: it would delay 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 object-type. 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 that are 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 (operating at or close to its maximum capacity). 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 command **ping** 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 users access to memory structures within the SGA.

See Also: Chapter 22, "The Dynamic Performance Views"
Oracle8 Concepts provides detailed information about each view.

SQL Trace Facility

SQL trace files record the SQL statements issued by a connected process and the resources used by these statements. In general, use the virtual tables to tune the instance, and use SQL trace file output to tune the applications.

See Also: Chapter 24, "The SQL Trace Facility and TKPROF"

Alert Log

Whenever something unexpected happens in an Oracle environment, it is worth checking 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 quite 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 the resource usages reported by a batch queue manager provides an excellent source of data for use in tuning.

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 file(s).

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 (stored procedures, constraints, and triggers) can be considered part of the same data population as client-side code, in this context. 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

A design or analysis dictionary can also be used to track the intended action and resource usage of the application system. Only where the application has been entirely produced by code generators, however, can the design dictionary provide all of the data which would otherwise have to be extracted from the programs and procedures themselves.

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 try to apply a solution that worked elsewhere. Normally, diagnoses such as these are purely subjective.

Tuning is much easier if a baseline exists, such as a capacity study performed for this application or (even better) data from this or another site running the same application with acceptable performance. The task is then to identify all differences between the two environments and attempt to bring them back into line.

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 tool for monitoring the performance of Oracle is the collection of dynamic performance views that Oracle provides to monitor your system. These views have names beginning with "V$", and this manual demonstrates their use in performance tuning. The database user SYS owns these views, and administrators can grant any database user access to them. Only some of these views are relevant to tuning your system.

See Also: Chapter 22, "The Dynamic Performance Views"
Oracle8 Concepts provides detailed information about each view.

Oracle and SNMP Support

The Simple Network Management Protocol (SNMP) enables users to write their own tools and applications. It is acknowledged as the standard, open protocol for heterogeneous management applications. Oracle SNMP support enables Oracle databases to be discovered on the network, identified, and monitored by any SNMP-based management application. Oracle supports several database management information bases (MIBs): the standard MIB for any database management system (independent of vendor), and Oracle-specific MIBs which contain Oracle-specific information. Some statistics mentioned in this manual are supported by these MIBs, and others are not. If a statistic mentioned in this manual can be obtained through SNMP, this fact is noted.

See Also: *Oracle SNMP Support Reference Guide*

EXPLAIN PLAN

EXPLAIN PLAN is a SQL statement that lists the access path determined by the query optimizer. Each plan has a row with ID = 0, which gives the statement type.

EXPLAIN PLAN results should be interpreted with some discretion. Just because a plan does not seem efficient on the surface does not necessarily mean that the statement will run slowly. Choose statements for tuning based upon their actual resource consumption, not upon a subjective view of their execution plan.

See Also: Chapter 23, "The EXPLAIN PLAN Command"

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 resource. With experience or with baselines available, you can gauge whether the resources used are reasonable, given the work accomplished.

See Also: Chapter 24, "The SQL Trace Facility and TKPROF"

Supported Scripts

Oracle provides many PL/SQL packages, thus a good number of SQL*Plus scripts that support instance tuning are available. Examples include UTLBSTAT.SQL and UTLESTAT; SQLUTLCHAIN.SQL, UTLDTREE.SQL, and UTLLOCKT.SQL.

These statistical scripts support instance management, allowing a history to be built up over time. They can be used for the following purposes:

- To remove the need to issue DDL each time statistics are gathered
- To separate data gathering from reporting, and to 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
- To report a number of indicative ratios you can check to determine whether the instance is adequately tuned
- To present LRU statistics from the buffer cache in a usable form

Application Registration

You can register with the database the name of an application 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.

See Also: Chapter 26, "Registering Applications"

Oracle Enterprise Manager Applications

This section describes Oracle Enterprise Manager and several of its most useful diagnostic and tuning tools. It covers:

- Introduction to Oracle Enterprise Manager
- Oracle Performance Manager
- Oracle TopSessions
- Oracle Trace
- Oracle Tablespace Manager
- Oracle Expert

Introduction to Oracle Enterprise Manager

Oracle Enterprise Manager is a major new infrastructure and tool set for managing 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 windows-based console for central administration and monitoring of Oracle databases
- Common services for event management, service discovery, security, and job creation / execution
- A server-side intelligent agent for monitoring events, running jobs, and communicating with the console
- Applications for administering Oracle databases for security, storage, backup, recovery, import, and software distribution

- Layered applications for managing replication, media, web, text, mobile agents, and other Oracle servers
- Optional products for Oracle monitoring and tuning, known as the Oracle Performance Pack

The Oracle Enterprise Manager Performance Pack is a set of windows-based applications built on the new Oracle Enterprise Manager systems management technology. These applications address many Oracle performance management areas, such as graphical monitoring, analysis, and automated tuning of Oracle databases.

Oracle Performance Manager

The Oracle Performance Manager captures, computes, and presents performance data that allows you to monitor key metrics required to use memory effectively, minimize disk I/O, and avoid resource contention. It provides a graphical, real-time view of Oracle performance metrics, and lets you drill down into a monitoring view for quick access to detailed data for performance problem solving. Oracle dynamic performance data is captured and displayed in real-time mode, and can be recorded for replay. 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, and can customize the rate of monitoring. You can also extend the system by defining charts for their own monitored sources (additional database performance data or application statistics).

The Oracle Performance Manager tracks real-time memory performance in several ways, providing data that can be put to use immediately for memory performance management. For example, the Parse Ratio Chart gives you a measure of the application's success at finding available parsed SQL in the database's library cache buffer, potentially indicating that shared pool memory allocation is insufficient. Monitor Charts can also be linked together, allowing you to drill down in a logical progression of analysis. For example, if you detect a performance problem with the Library Cache Hit Ratio, you can drill down to the Library Cache Details Chart. Other memory monitoring charts include: Data Dictionary Cache Hit Ratio, Memory Allocated, and Sort Hit Ratio, to name a few.

Using the Oracle Performance Manager, you can chart virtually any data in your database, whether this data is database performance related or data from your business application tables that you want to chart. Oracle Monitor provides dialog boxes for entering the SQL to retrieve the data, for defining operations to be performed on the data, and for selecting the type of chart best suited to graphically display the data. This ability to define your own charts can be combined with the power of Oracle Trace to create custom charts for monitoring application perfor-

mance, application audit trails, or business transaction data. Using Oracle Trace in this way will be discussed in more detail later.

Performance problems detected by using the Oracle Monitor can be corrected by using other Oracle Enterprise Manager applications. For example, memory management problems that arise from inappropriate buffer sizes can be corrected easily using the Oracle Instance Manager application to reset buffer size parameters. Likewise, you can address I/O or contention problems by using the Oracle Storage Manager application to reset storage parameters, or the Oracle Tablespace Manager application to further analyze the problem and defragment tables if necessary.

In addition, when you detect performance problems through the Oracle Monitor you can obtain a far greater degree of detail through two other Performance Pack applications: Oracle TopSessions and Oracle Trace. Ultimately, you can elect to have a detailed tuning analysis conducted by the Oracle Expert automated performance tuning application. Oracle Expert produces recommendations and scripts for improving the performance of the database being monitored.

Oracle TopSessions

Often a DBA needs more information than provided by general database monitoring. For example, a DBA using the Oracle Performance Manager may detect a file I/O problem. In order to solve the problem quickly, it would be helpful to know which particular user sessions are causing the greatest I/O activity.

Oracle TopSessions provides the DBA with a focused view of performance activity for the top set of Oracle sessions at any given time. Oracle TopSessions extracts and analyzes sample Oracle dynamic performance data, automatically determining the top Oracle user sessions based on a specific selection criterion, such as file I/O activity. Using Oracle TopSessions, the DBA can quickly detect which user sessions are causing the greatest file I/O activity and require further investigation.

Oracle TopSessions provides two views of session data: an Overview of a select number of top sessions, and a Session Details view. The application starts with an overview of the top ten sessions connected to the database instance, with an initial default sort based on session PGA memory usage. The data displayed in the initial overview includes items such as session ID, node, application, username, last session command executed, and the status of the session (idle, active, blocked, or killed). You can then customize the display by changing the number of sessions to be monitored and selecting the type of statistical filtering and sorting to be done for the Overview display of monitored sessions.

The Session Details display allows you to drill down into a particular session, providing pages for detailed displays of general session information, session statistics,

cursors, and locks. The Session Details General page expands the information provided in the Overview display, adding information such as identifiers for the schema, SQL, deadfalls, rows, and blocks as applicable. The Statistics page displays detailed performance statistics for the session that are captured from the V$SESSTAT view. The Cursors page provides information on all shared cursors for the session, including SQL statements and EXPLAIN PLAN output. You can display the session's currently executing SQL statements, or all SQL statements that have been and will be executed for the session. The Session Details Locks page displays information about the database locks held or requested by session.

When monitoring multiple instances, you can open as many Oracle TopSessions displays as necessary. The information displayed in Oracle TopSessions is static until refreshed. Oracle TopSessions allows you to determine whether the refresh should be manual or automatic, and the frequency of automatic refresh.

Oracle Trace

Most data used in performance monitoring applications is collected based on sampling methodologies. For example, Oracle Performance Manager and Oracle TopSessions use this technique by periodically collecting data from the Oracle dynamic performance views.

Oracle Trace provides a new data collection methodology that goes a significant step further than sampling techniques. Oracle Trace collects performance data for each and every occurrence of key events in an application being monitored. It provides an entire census of performance data, rather than a sample of data, for a software application or database event. This allows performance problems detected through sampling techniques to be pinpointed to specific occurrences of a software product's execution.

Oracle Trace collects performance data for predefined events in products such as the Oracle Server, Net8, and any other Oracle or third-party application that has been programmed with the Oracle Trace data collection API. An Oracle Trace "event" is an occurrence within the software product containing the Oracle Trace API calls. For example, specific events have been identified in the Oracle Server, such as a SQL parse, execute, and fetch. These events have been delimited with API calls, which are invoked when the event occurs during a scheduled Oracle Trace collection for the Oracle Server. Another example of an event to be monitored for performance data would be a transaction in an application, such as a deposit in a banking application. Any product can be programmed with Oracle Trace API calls for event-based data collection.

The type of performance data collected for events includes extensive resource utilization data, such as CPU time, memory usage, and page faults, as well as perfor-

mance data specific to the product being monitored. For example, user and form identification data would likely be collected for business application events, in addition to resource utilization data for those events. In addition, Oracle Trace provides the unique capability to correlate the performance data collected across any end-to-end client/server application containing Oracle Trace instrumented products. Performance can be tracked across multiple products involved in the enterprise transaction, allowing the application developer, DBA, or systems manager to easily identify the source of performance problems.

From a DBA's perspective, the value of Oracle Trace is embodied in the products that use Oracle Trace data for analysis and performance management. DBAs and other users do not have to instrument an application in order to use Oracle Trace. Rather, the majority of users will employ Oracle Trace to collect data for a product that already contains the API calls, and will likely use the collected data in some other tool that performs monitoring or analysis. For example, Oracle Server and Net8 contain Oracle Trace API calls for event data collection. An Oracle Trace user can schedule a collection of Oracle Trace data for either of these products, format the data and review it in reports. In addition, Oracle Trace data for Oracle Server can be imported into the Oracle Expert database tuning application, where it will be automatically analyzed for Oracle server tuning.

See Also: Chapter 25, "Using Oracle Trace"

Oracle Tablespace Manager

If you suspect database performance problems due to tablespace disorganization, you can use the Oracle Tablespace Manager to investigate and correct structure problems. The Oracle Tablespace Manager consists of two major features: a Tablespace Viewer and a tablespace defragmentation function.

The Tablespace Viewer 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. You can display all segments for a tablespace or all segments for a datafile. The Tablespace Viewer also provides a map of the organization of a tablespace's segments. This map graphically displays the sequential allocation of space for segment extents within a selected tablespace or datafile. For example, a table segment may consist of three extents, all of which are physically separated by other segment extents. The map will highlight the locations of the three extents within the tablespace or datafile. It will also show the amount of free space available for each segment. In this way, the Tablespace Viewer map provides an easy bird's-eye view of tablespace fragmentation.

When tablespace fragmentation is detected, you can use the Oracle Tablespace Manager defragmentation feature to correct the problem automatically. You can select a table for defragmentation from the list of table segments. The defragmentation process uses the Oracle export/import functions on the target table, and ensures that all rows, indexes, constraints and grants remain intact. Before the table export occurs, you are presented with a dialog box for modifying the storage parameters for the selected table if desired. The new parameters are then used in the re-creation of the defragmented table. You also have the option of compressing the table's extents into one large initial extent.

In addition to managing fragmentation, a DBA must watch for opportunities to use available database resources more effectively. A database incurring lots of updates and deletes will develop empty data blocks—pockets of free space that are too small for new extents. The Tablespace Viewer allows you to visually identify free blocks. If these free blocks are adjacent, they can be joined automatically using the Oracle Tablespace Manager's coalesce feature. In this way they will become more useful space for future extents.

Oracle Expert

Oracle Expert provides automated performance tuning. Performance problems detected by the Oracle Performance Manager, Oracle TopSessions, and Oracle Trace can be analyzed and solved with Oracle Expert. Oracle Expert automates the process of collecting and analyzing data, and contains rules that provide database tuning recommendations, implementation scripts, and reports. Oracle Expert monitors several factors in the database environment and provides tuning recommendations in three major tuning categories:

- Access method tuning
- Instance parameter tuning
- Database structure sizing and placement

You can select a single tuning category for focused tuning or multiple categories for more comprehensive tuning. Tuning can also be focused on a specific portion of the database, such as a table or index. You can graphically view and edit the data collected by Oracle Expert, including database workload, systems environment, database schema, instance parameters and statistics, tablespace data, and so forth. You can also modify Oracle Expert's rule values, for example, increasing or decreasing a rule's decision threshold. This powerful feature allows you to play a part in the analysis and recommendations produced by Oracle Expert. You can employ this capability to customize the collected data in order to test various tuning scenarios and to influence the final results.

After Oracle Expert has analyzed the data, you can review the recommendations, including selectively viewing the detailed analysis. You can choose to accept specific recommendations before generating the recommended implementation files, which generally consist of new instance parameter files and implementation scripts. You have full control over the implementation process: invoke the implementation files when you are ready, and they will automatically implement the changes you have accepted. Oracle Expert also produces a series of reports to document the data and analysis behind the recommendations. These reports provide extensive documentation for the database and tuning process. For the less experienced DBA, they can be a valuable education in the factors that drive database performance.

In summary, Oracle Expert, along with the other Performance Pack applications, provides you with a useful set of tools for monitoring and tuning Oracle databases.

Oracle Parallel Server Management

Oracle Parallel Server Management (OPSM) is a comprehensive and integrated system management solution for the Oracle Parallel Server. OPSM allows you to manage multi-instance databases running in heterogeneous environments through an open client-server architecture.

In addition to managing parallel databases, OPSM allows you to schedule jobs, perform event management, monitor performance, and obtain statistics to tune parallel databases.

For more information about OPSM, 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 over the course of several years. Such tools might include free space monitors, to determine whether tables have enough space to be able to extend; lock monitoring tools; schema description scripts to show tables and all associated indexes; and tools to show default and temporary tablespaces per user. You can integrate such programs with Oracle by setting them to run automatically.

Part II

Designing and Developing for Performance

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

- Chapter 5, “Evaluating Your System’s Performance Characteristics”
- Chapter 6, “Designing Data Warehouse Applications”

5

Evaluating Your System's Performance Characteristics

This chapter describes the various types of application that use Oracle databases and the suggested approaches and features available when designing each type. Topics in this chapter include

- Types of Application
- Oracle Configurations

Types of Application

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

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

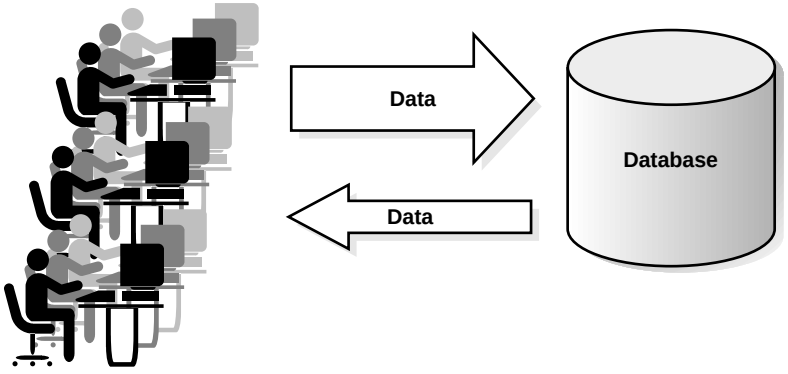
See Also: *Oracle8 Concepts*, *Oracle8 Application Developer's Guide*, and *Oracle8 Administrator's Guide* for more information on these topics and how to implement them in your system.

Online Transaction Processing (OLTP)

Online transaction processing (OLTP) applications are high-throughput, insert/update-intensive systems. These systems are characterized by constantly growing large 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 an OLTP system are availability (sometimes 7 day/24 hour availability); speed (throughput); concurrency; and recoverability.

Figure 5–1 illustrates the interaction between an OLTP application and an Oracle Server.

Figure 5–1 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 issues are crucial in tuning an OLTP system:

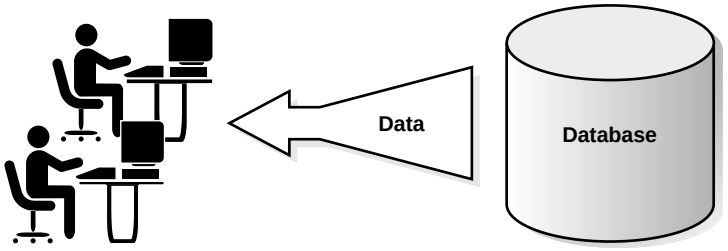
- rollback segments
- indexes, clusters, and hashing
- discrete transactions
- data block size
- dynamic allocation of space to tables and rollback segments
- transaction processing monitors and the multithreaded server
- the shared pool
- well-tuned SQL statements
- integrity constraints
- client / server architecture
- dynamically changeable initialization parameters
- procedures, packages, and functions

See Also: *Oracle8 Concepts* and *Oracle8 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 distill large amounts of information into understandable reports. Typically, decision support applications perform queries on the large amount of data gathered from OLTP applications. Decision makers in an organization use these applications to determine what strategies the organization should take, based on the available information. Figure 5–2 illustrates the interaction between a decision support application and an Oracle Server.

Figure 5–2 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 to 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 you design a decision support system, you must ensure that queries on large amounts of data can be performed within a reasonable time frame. Decision makers often need reports on a daily basis, so you may need to guarantee that the report can complete 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 tuning a decision support system:

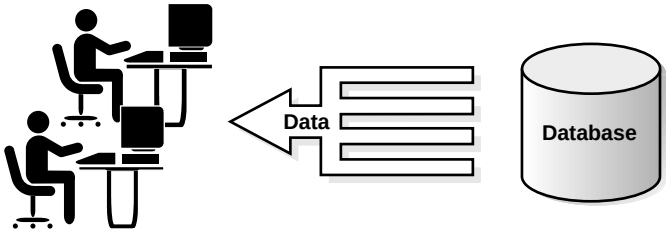
- Indexes (B*-tree and bitmap)
- Clusters, hashing
- Data block size
- Parallel execution

- 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, which enables multiple processes to work together simultaneously to process a single SQL statement. By dividing the work necessary to process a statement among multiple server processes, the Oracle Server can process the statement more quickly than if only a single server process processed it.

Figure 5–3 illustrates the parallel execution feature of the Oracle Server.

Figure 5–3 Parallel Query Processing



Parallel execution can dramatically improve performance for data-intensive operations associated with decision support applications or very large database environments. Symmetric multiprocessing (SMP), clustered, or massively parallel systems gain the largest performance benefits from parallel execution, because the operation can be effectively split 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, you need to alleviate the system's load before using parallel execution to improve performance.

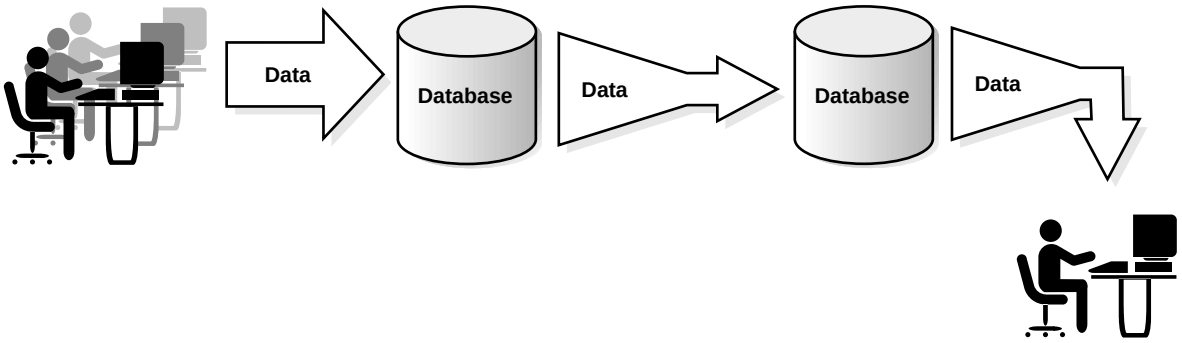
See Also: Chapter 6, "Designing Data Warehouse Applications" for an introduction to Oracle data warehousing functionality.
Chapter 19, "Tuning Parallel Execution", introduces performance aspects of parallel execution.
Oracle8 Concepts provides general information about parallel execution.

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 5–4 illustrates multiple configurations and applications accessing an Oracle Server.

Figure 5–4 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 a large amount of data from daily purchases, 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 cause performance problems for both parts of the system. To solve this problem, 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 slightly compromises the goal of accuracy for the data warehousing application (the data is copied only once per day), but the trade-off is significantly better performance from both systems.

For hybrid systems you must 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 to you. The basic configurations are:

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

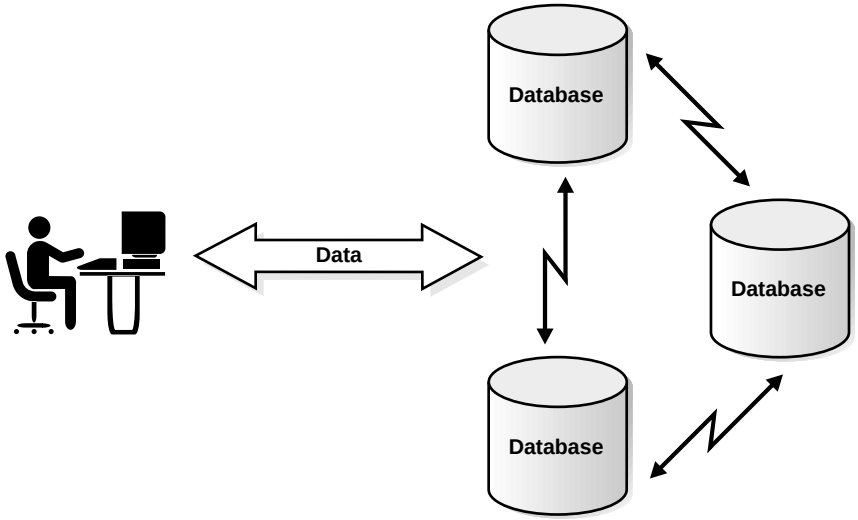
Depending on your application and your operating system, each of these or a combination of these configurations will best suit your needs.

Distributed Systems

Distributed applications involve spreading data over multiple databases on multiple machines. Several smaller server machines can be cheaper and more flexible than one large, centrally located server. This configuration takes advantage of small, powerful server machines and cheaper connectivity options. Distributed systems allow you to have data physically located at several sites, and each site can transparently access all of the data.

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

Figure 5–5 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 the clerks perform local operations on an order-entry system. The local orders are forwarded each day to the central shipping department. While the inventory and shipping departments are centrally located, the clerks are spread across the country for the convenience of the customers.

The key goals of a distributed database system are availability, accuracy, concurrency, and recoverability. When you design a distributed database 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, and 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:

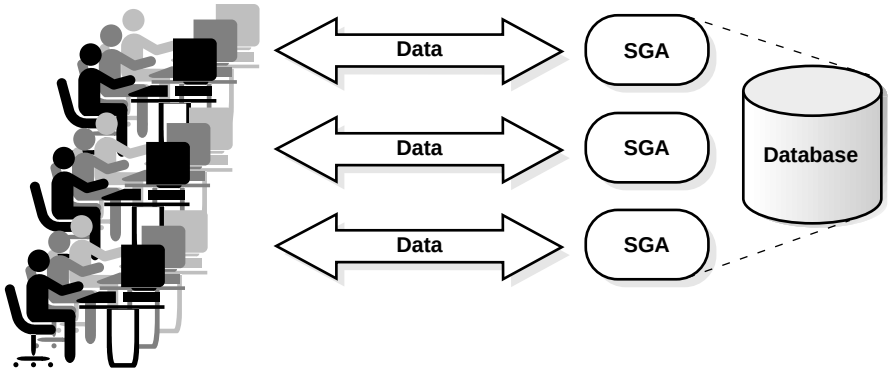
- Network configuration
- Distributed database design
- Symmetric replication
- Table snapshots and snapshot logs
- Procedures, packages, and functions

See Also: *Oracle8 Distributed Database Systems* and *Oracle8 Replication* Chapter 9, "Tuning Distributed Queries"

The Oracle Parallel Server

The Oracle Parallel Server is available on clustered or massively parallel systems. A parallel server allows multiple machines to have separate instances all accessing the same database. This configuration greatly enhances data throughput. Figure 5–6 illustrates the Parallel Server option of the Oracle Server.

Figure 5–6 An Oracle Parallel Server



When you configure a system with the Parallel Server option, your key concern is data contention among the various nodes. Each node that requires updatable data must first obtain a lock on that data to ensure data consistency. If multiple nodes all want to access the same data, that data must first be written to disk, and then the next node can obtain the lock. This type of contention can significantly slow a parallel server; on such systems data must be effectively partitioned among the various nodes for optimal performance. (Note that read-only data can be efficiently shared across the parallel server without the problem of lock contention.)

See Also: *Oracle8 Parallel Server Concepts & Administration*

Client/Server Configurations

Client / server architecture distributes 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 Architecture" on page 13-10 for information about multi-tier systems.

6

Designing Data Warehouse Applications

This chapter introduces integrated Oracle8 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

- Introduction
- Features for Building a Data Warehouse
- Features for Querying a Data Warehouse
- Backup and Recovery of the Data Warehouse

Introduction

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 resource 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 be performed on 100 gigabytes of data. At a processing speed of 0.2 G to 2 G of data per hour per CPU, a single CPU might need from 2 days to more than 2 weeks to perform a task. With more than a single gigabyte of data (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 10 hours. By contrast, using parallel CREATE TABLE . . . AS SELECT on 10 CPUs might take only 1 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.

Features for Building a Data Warehouse

This section outlines Oracle8 features useful for building a data warehouse. It includes:

- Parallel CREATE TABLE . . . AS SELECT
- Parallel Index Creation
- Fast Full Index Scan
- Partitioned Tables
- ANALYZE Command
- Parallel Load

See Also: *Oracle8 Concepts* and *Oracle8 SQL Reference*

Parallel CREATE TABLE . . . AS SELECT

The ability to CREATE TABLE . . . AS SELECT 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 19-51
Oracle8 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 afterward.

See Also: "Creating Indexes in Parallel" on page 19-53
Chapter 19, “Tuning Parallel Execution”

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: "Fast Full Index Scan" on page 10-9
"INDEX_FFS" on page 8-22

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 Oracle8 partitioned tables should be used rather than partition views. Please see *Oracle8 Migration* for instructions on migrating from partition views to partitioned tables.

See Also: *Oracle8 Concepts* for information about partitioned tables
"Partitioning Data" on page 19-23

ANALYZE Command

You can use the ANALYZE command 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. Note that statistics can be either computed or estimated, depending on the amount of overhead you are willing to allow for this purpose.

See Also:
"Step 3: Analyzing Data" on page 19-36
Oracle8 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 19, "Tuning Parallel Execution", especially
"Using Parallel Load" on page 19-25
Oracle8 Utilities for a description of SQL Loader conventional and direct path loads.

Features for Querying a Data Warehouse

This section summarizes Oracle8 features useful for querying a data warehouse. It includes:

- Oracle Parallel Server Option
- Parallel-Aware Optimizer
- Parallel Execution
- Bitmap Indexes
- Star Queries
- Star Transformation

Oracle Parallel Server Option

The Oracle Parallel Server option provides benefits important to both OLTP and data warehousing applications:

- application failover
- scalable performance
- load balancing
- multiuser 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 query 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 query 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 query on clusters and MPPs, in a mostly query environment the overhead on the distributed lock manager is minimal.

See Also: *Oracle8 Parallel Server Concepts & Administration*

Parallel-Aware Optimizer

Knowledge about parallelism is incorporated into the Oracle8 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 query.

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 19-4

Parallel Execution

The Oracle8 provides for improved performance through use of parallel execution.

Parallel execution enables multiple processes to work together simultaneously to process a single query or DML statement. By dividing the task among multiple server processes, Oracle can execute the operation more quickly than if only one server process were used.

Parallel execution can dramatically improve performance for data-intensive data warehousing operations. It helps systems scale in performance when adding hardware resources. The greatest performance benefits are on symmetric multiprocessing (SMP), clustered, or massively parallel systems where query processing can be effectively spread out among many CPUs on a single system.

See Also: Chapter 19, “Tuning Parallel Execution”
Oracle8 Concepts for conceptual background 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, which makes it very fast and space efficient. Bitmap indexing benefits data warehousing applications that have large amounts of data and ad hoc queries, but a low level of concurrent transactions. It provides reduced response time for many kinds of ad hoc queries; considerably reduced space usage compared to other indexing techniques; and dramatic performance gains even on very low end hardware. Bitmap indexes can be created in parallel and are completely integrated with cost-based optimization.

See Also: "Using Bitmap Indexes" on page 10-13

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, simply ANALYZE your tables and be sure that the OPTIMIZER_MODE initialization parameter is set to its default value of CHOOSE.

See Also: *Oracle8 Concepts* regarding optimization of star queries
"STAR" on page 8-24 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: *Oracle8 Concepts* for a full discussion of star transformation.
Oracle8 Reference describes the STAR_TRANSFORMATION_ENABLED initialization parameter.
"STAR_TRANSFORMATION" on page 8-35 explains how to use this hint.

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. Oracle8 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 the *Oracle8 Administrator's Guide* and *Oracle8 SQL Reference.*
"[NO]LOGGING Option" on page 19-35

Part III

Optimizing Database Operations

Part 3 discusses how to tune your database and the various methods you use to access data for optimal database performance. The chapters in Part 3 are:

- Chapter 7, “Tuning Database Operations”
- Chapter 8, “Optimization Modes and Hints”
- Chapter 9, “Tuning Distributed Queries”
- Chapter 10, “Data Access Methods”
- Chapter 11, “Oracle8 Transaction Modes”
- Chapter 12, “Managing SQL and Shared PL/SQL Areas”

7

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:

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

Tuning Goals

This section introduces:

- Tuning a Serial SQL Statement
- Tuning Parallel Operations
- Tuning OLTP Applications
- Tuning Data Warehouse 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 has a high correlation with parallel operations and high data volume.
- OLTP applications have a high correlation with serial operations and high transaction volume.

As a result, these applications have contrasting goals for tuning.

Table 7–1 Contrasting Goals for Tuning

Tuning Situation	Goal
Serial SQL Statement	Minimize resource utilization 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 follows:

To minimize resource utilization by the operation being performed.

You can explore alternative syntax for SQL statements without actually modifying your application. Simply use the EXPLAIN PLAN command with the alternative statement that you are considering and compare its execution plan and cost with that of the existing statement. 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: Chapter 23, “The EXPLAIN PLAN Command”
"Approaches to SQL Statement Tuning" on page 7-6

Tuning Parallel Operations

The goal of tuning parallel operations can be stated thus:

To maximize throughput for the given hardware.

If you have a powerful system and a massive, high-priority SQL statement to run, you want to parallelize the statement so that it utilizes 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 up the rows so they are not all done by a single process.

See Also: Chapter 19, "Tuning Parallel Execution"
Oracle8 Concepts, for basic principles of parallel execution

Tuning OLTP Applications

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

SQL and Shared PL/SQL

To minimize parsing, use bind variables in SQL statements within OLTP applications. In this way all users will be able to share the same SQL statements, and fewer resources will be required for parsing.

Transaction Modes

Sophisticated users can use discrete transactions if performance is of the utmost importance, and if they 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 11, "Oracle8 Transaction Modes"

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 12, "Managing SQL and Shared PL/SQL Areas"

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 so They Use Less Resources

Step 1: Find the Statements that Consume the Most Resources

Focus your tuning efforts on those statements where the benefit of tuning will demonstrably exceed 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, which gives the session and SQL statement associated with a temporary segment.

The statements that have 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 24, “The SQL Trace Facility and TKPROF”
Chapter 25, “Using Oracle Trace”
Oracle8 Reference for more information about dynamic performance views

Step 2: Tune These Statements so They Use Less Resources

Remember that application design is fundamental to performance. No amount of SQL statement tuning can make up for inefficient design. If you encounter stumbling blocks in SQL statement tuning, 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 less resources when it is used.
- Use the statement less frequently.

Statements may use the most 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 13, "Tuning CPU Resources"
Chapter 15, "Tuning I/O"

Approaches to SQL Statement Tuning

This section describes three strategies you can use to speed up SQL statements:

- Restructure the Indexes
- Restructure the Statement
- Restructure the Data

Note: These guidelines 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

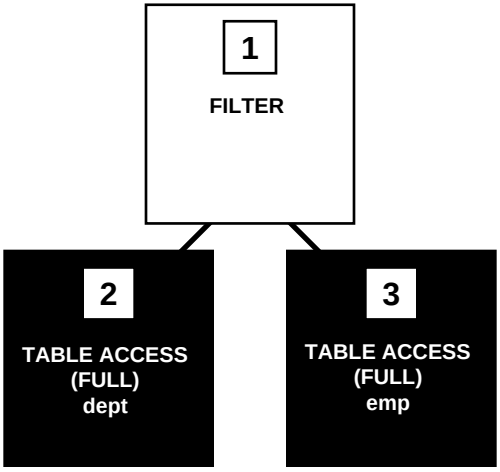
Because SQL is a flexible language, more than one SQL statement may meet the needs of your application. Although two SQL statements may produce the same result, Oracle may process one faster than the other. You can use the results of the EXPLAIN PLAN statement to compare the execution plans and costs 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 7–1 Execution Plan with Two Full Table Scans

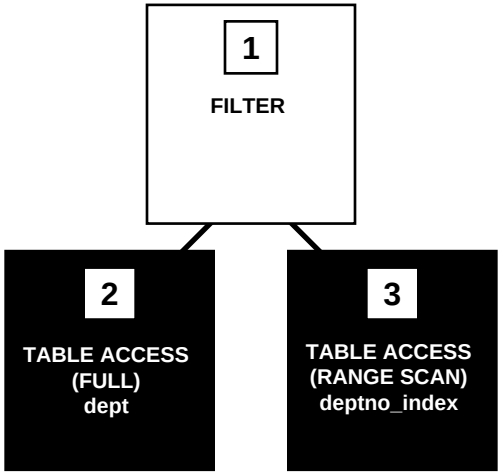


Step 3 of the output indicates that Oracle executes this statement by performing a full table scan of the EMP table despite the index on the DEPTNO column. This full table scan can be a time-consuming operation. Oracle does not use the index because the subquery that searches the EMP table does not have 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 7–2 Execution Plan with a Full Table Scan and an Index Scan



See Also: The optimizer chapter in *Oracle8 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 impact 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 doing 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 stuff
  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-Mode 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 stuff 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 */ stuff
  FROM tables
 WHERE ...
   AND somecolumn between :loval and :hival
   AMD (:hival != 'ALL' and :loval != 'ALL')
UNION ALL
SELECT /* Change this half of union all if other half changes. */ stuff
  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';
```

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: *Oracle8 SQL Reference* for more information.

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
 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.

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 procedurally either in PL/SQL within a stored procedure, or within the user interface code.

When considering a cross-network join, note that 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.

8

Optimization Modes and Hints

This chapter explains when to use the available optimization modes and how to use hints to enhance Oracle performance.

Topics include:

- Using Cost-Based Optimization
- Using Rule-Based Optimization
- Introduction to Hints
- How to Specify Hints
- Hints for Optimization Approaches and Goals
- Hints for Access Methods
- Hints for Join Orders
- Hints for Join Operations
- Hints for Parallel Execution
- Additional Hints
- Using Hints with Views

See Also: *Oracle8 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
- How to Use the Cost-Based Approach
- Using Histograms for Nonuniformly Distributed Data
- Generating Statistics
- Choosing a Goal for the Cost-Based Approach
- Parameters that Affect Cost-Based Optimization Plans
- Tips for Using the Cost-Based Approach

When to Use the Cost-Based Approach

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

The following features are available *only* with cost-based optimization; you *must* analyze your tables to get good plans:

- partitioned tables
- index-only tables
- reverse indexes
- parallel query
- star transformation
- star join

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, especially for large queries with multiple joins or multiple indexes. The cost-based approach also improves productivity by eliminating the need to tune your SQL statements yourself. Finally, many Oracle performance features are available only through the cost-based approach.

Cost-based optimization must be used for efficient star query performance. Similarly, it must be used with hash joins and histograms. Cost-based optimization is always used with parallel query and with partitioned tables. You must use the ANALYZE command in order to keep the statistics current.

How to Use the Cost-Based Approach

To use cost-based optimization for a statement, first collect statistics for the tables accessed by the statement. Then enable cost-based optimization in one of these ways:

- 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 . . . 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 that the plan chosen for the full database will be the same as that chosen for the prototype.

Using Histograms for Nonuniformly Distributed Data

For nonuniformly distributed data, you should create histograms describing the data distribution of particular columns. For this type of data, histograms enable the cost-based optimization approach to accurately guess the cost of executing a particular statement. For data that is uniformly distributed, the optimizer does not need histograms to accurately estimate the selectivity of a query.

How to Use Histograms

Create histograms on columns that are frequently used in WHERE clauses of queries and have a highly skewed data distribution. You create a histogram by using the ANALYZE TABLE command. For example, if you want to create a 10-bucket histogram on the SAL column of the EMP table, issue the following statement:

```
ANALYZE TABLE emp COMPUTE STATISTICS FOR COLUMNS sal SIZE 10;
```

The SIZE keyword states the maximum number of buckets for the histogram. You would create a histogram on the SAL column if there were an unusual number of employees with the same salary and few employees with other salaries.

See Also: *Oracle8 SQL Reference* for more information about the ANALYZE command and its options.

Choosing the Number of Buckets for a Histogram

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

If the number of frequently occurring distinct values in a column is relatively small, then it is useful to 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 through the following 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: *Oracle8 Concepts* for column descriptions of data dictionary views, as well as histogram use and restrictions.

Generating Statistics

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

Oracle can generate statistics using these techniques:

- estimation based on random data sampling
- exact computation

Use estimation, rather than computation, unless you think you need exact values, because:

- 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 of the table.
- Estimation is often much faster than computation, especially for large tables, because estimation never scans the entire table.

To perform a 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 of the rows in the requested sample of the table.

Because of the time and space required for the computation of table statistics, it is usually best to perform an estimation for tables and clusters. For indexes, computation does not take up as much time or space, so it is best to perform a 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 with the new ones. Oracle invalidates any currently parsed SQL statements that access any of the analyzed objects. When such a statement is next executed, 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 when they are next parsed.

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

You can generate statistics with the ANALYZE command.

Example: This example generates statistics for the EMP table and its indexes:

```
ANALYZE TABLE emp
   ESTIMATE STATISTICS;
```

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 is more likely to result in an index scan or a nested loops join.

For example, consider a join statement that can be executed 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 the goal is best throughput, the optimizer is more likely to choose a sort-merge join. If the goal is best 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...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;
```

Parameters that Affect 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, COMPLEX_VIEW_MERGING, PUSH_JOIN_PREDICATE, FAST_FULL_SCAN_ENABLED
OPTIMIZER_MODE	As 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. Use OPTIMIZER_MODE option of 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 value causes hash join costs to be cheaper, giving more hash joins.
SORT_AREA_SIZE	Large value causes sort costs to be cheaper, giving more sort merge joins.
DB_FILE_MULTIBLOCK_READ_COUNT	Large value gives cheaper table scan cost and favors table scans over indexes.
COMPLEX_VIEW_MERGING	Controls complex view merging.
PUSH_JOIN_PREDICATE	Enables the optimizer to evaluate whether or not to push individual join predicates into the view query block.

The following parameters often need to be set in a data warehousing application:

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.
SORT_DIRECT_WRITES	Gives lower sort costs and more sort merge joins.

The following parameters rarely need to be changed:

HASH_MULTIBLOCK_IO_COUNT	Larger value causes hash join costs to be cheaper, giving more hash joins.
SORT_WRITE_BUFFER_SIZE	Large value causes sort costs to be cheaper, giving more sort merge 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.

Note: The following sort parameters can be modified using ALTER SESSION ... SET or ALTER SYSTEM ... SET DEFERRED:

SORT_AREA_SIZE
SORT_AREA_RETAINED_SIZE
SORT_DIRECT_WRITES
SORT_WRITE_BUFFERS
SORT_WRITE_BUFFER_SIZE
SORT_READ_FAC

See Also: *Oracle8 Reference* for complete information about each parameter.

COMPLEX_VIEW_MERGING

Recommended value: default

When set to FALSE, this parameter causes complex views or subqueries to be evaluated before the referencing query. In this case, you can cause a view to be merged on a per-query basis by using the MERGE hint.

When set to TRUE, this parameter causes complex views or subqueries to be merged. In this case, you can use the NO_MERGE hint within the view to prevent one particular view from being merged. Alternatively, you can use the NO_MERGE hint in the surrounding query, and specify the name of the view that should not be merged.

PUSH_JOIN_PREDICATE

Recommended value: TRUE

When this parameter is set to TRUE, the optimizer can evaluate, on a cost basis, whether or not to push individual join predicates into the view query block. This can enable more efficient access path and join methods, such as transforming hash joins into nested loop joins, and full table scans to index scans.

If PUSH_JOIN_PREDICATE is TRUE, you can use the NO_PUSH_JOIN_PRED hint to prevent pushing join predicates into the view.

If PUSH_JOIN_PREDICATE is FALSE, you can use the PUSH_JOIN_PRED hint to force pushing of a join predicate into the view.

Tips for Using the Cost-Based Approach

The cost-based optimization approach assumes that a query will be executed on a multiuser system with a fairly low buffer cache hit rate. Thus a plan selected by cost-based optimization may not be the best plan for a single user system with a large buffer cache. 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 multiuser system.

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

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

Using Rule-Based Optimization

Rule-based optimization is supported in Oracle8, but you are advised to write any new applications using cost-based optimization. Cost-based optimization should be used for new applications and for data warehousing applications, because it supports new and enhanced features. Much of the functionality in Oracle8 (such as hash joins, improved star query processing, and histograms) is available only through cost-based optimization.

If you have developed existing 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 Oracle8.

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 the rule-based approach will not be available in future versions of Oracle.

If you are using an application provided by a third-party vendor, check with the vendor 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 them or by setting either the value of the OPTIMIZER_MODE initialization parameter or the OPTIMIZER_MODE option of the ALTER SESSION command 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.

Introduction to Hints

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

Hints are suggestions that you give the optimizer for optimizing a SQL statement. 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, however, that the use of hints involves extra code that must also be managed, checked, controlled.

How to Specify 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:

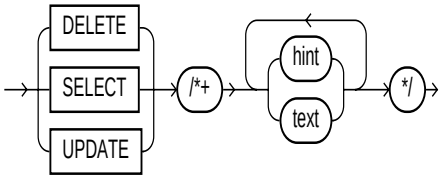
- a simple SELECT, UPDATE, or DELETE statement
- a parent statement or subquery of a complex statement
- a part of a compound query

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 this first component query apply only to its optimization, not to the optimization of the second component query.

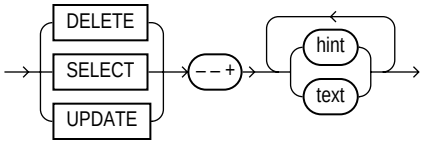
You can send hints for a SQL statement to the optimizer by enclosing them in a comment within the statement.

See Also: For more information on comments, see *Oracle8 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 also 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.

The optimizer recognizes hints only when using the cost-based approach. If you include any 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, between the goals of best throughput and best response time.

- ALL_ROWS
- FIRST_ROWS
- CHOOSE
- RULE

If a SQL statement contains a hint that specifies 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 command.

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).

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.

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
- group 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.

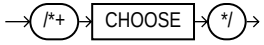
If you specify either the ALL_ROWS or FIRST_ROWS hint in a SQL statement and the data dictionary contains no statistics about any of the tables accessed by the statement, the optimizer uses default statistical values (such as allocated storage for

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 command; therefore, you should use the ANALYZE command 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 approach and the cost-based approach for a SQL statement based on the presence of statistics for the tables accessed by the statement. If the data dictionary contains 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 contains no statistics for any of these tables, the optimizer uses the rule-based approach.

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 any other hints specified for the statement block.

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
- INDEX
- INDEX_ASC
- INDEX_COMBINE
- INDEX_DESC
- INDEX_FFS
- MERGE_AJ
- AND_EQUAL
- USE_CONCAT

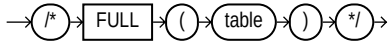
Specifying one of these hints causes the optimizer to choose the specified access path only if the access path is available based on the existence of an index or cluster and the syntactic constructs of the SQL statement. If a hint specifies an unavailable access path, the optimizer ignores it.

You must specify the table to be accessed exactly as it appears in the statement. If the statement uses an alias for the table, 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.

FULL

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

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.

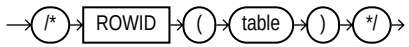
For example, Oracle performs a full table scan on the ACCOUNTS table to execute this statement, even if there is an index on the ACCNO column that is made available by the condition in the WHERE clause:

```
SELECT /*+ FULL(a) Don't use the index on ACCNO */ accno, bal
  FROM accounts a
 WHERE accno = 7086854;
```

Note: Because the ACCOUNTS table has an 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:

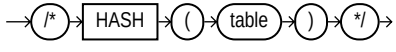
where *table* specifies the name or alias of the table to be accessed by a cluster scan.

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

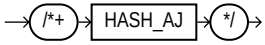
The HASH hint explicitly chooses a hash scan to access the specified table. It applies only to tables stored in a cluster. The syntax of the HASH hint is:



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

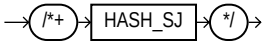
HASH_AJ

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:



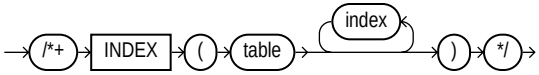
HASH_SJ

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. 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, which selects the name, height, and weight of all male patients in a hospital:

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

Assume that there is an index on the SEX column and that this column contains the values M and F. If there are equal numbers of male and female patients in the hospital, 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.

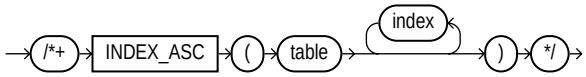
If you know that the value in the WHERE clause of your query appears in a very small percentage of the rows, you can use the INDEX hint to force the optimizer to choose an index scan. In this statement, the INDEX hint explicitly chooses an index scan on the SEX_INDEX, the index on the SEX column:

```
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.

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:

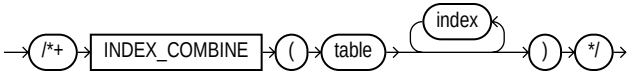


Each parameter serves the same purpose as in the INDEX hint.

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 currently specify anything more than the INDEX hint. However, you may want to use the INDEX_ASC hint to specify ascending range scans explicitly, should the default behavior change.

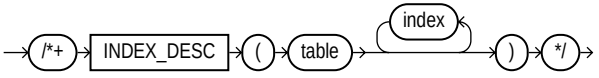
INDEX_COMBINE

If no indexes are given as arguments for the INDEX_COMBINE hint, the optimizer will use on the table whatever Boolean combination of bitmap indexes has the best cost estimate. If certain indexes are given as arguments, the optimizer will try to use some Boolean combination of those particular bitmap indexes. The syntax of INDEX_COMBINE is:



INDEX_DESC

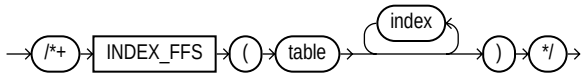
The INDEX_DESC hint explicitly chooses an index scan for the specified table. If the statement uses an index range scan, Oracle scans the index entries in descending order of their indexed values. Syntax of the INDEX_DESC hint is:



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.

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: "Fast Full Index Scan" on page 10-9

MERGE_AJ

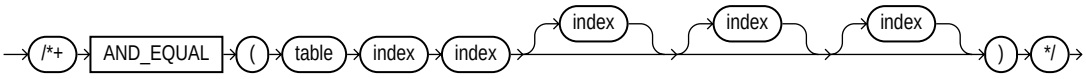
The MERGE_AJ hint transforms a NOT IN subquery into a merge anti-join to access the specified table. The syntax of the MERGE_AJ hint is:

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.

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.

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.

Syntax of this hint is:

Usually, if you analyze the tables the optimizer will choose an efficient star plan. You can also use hints to improve the plan. The most precise method is to order the tables in the FROM clause in the order of the keys in the index, with the large table last. Then use the following hints:

```
/*+ ORDERED USE_NL(facts) INDEX(facts fact_concat) */
```

Where “facts” is the table and “fact_concat” is the index. A more general method is to use the STAR hint.

See Also: *Oracle8 Concepts* for more information about star plans.

Hints for Join Operations

Each hint described in this section suggests a join operation for a table.

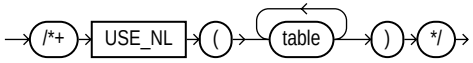
- USE_NL
- USE_MERGE
- USE_HASH
- USE_HASH
- DRIVING_SITE

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.

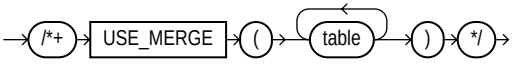
However, you may want to optimize the statement for best response time, or the minimal elapsed time necessary to return the first row selected by the query, rather than best throughput. If so, you can force the optimizer to choose a nested loops join by using the USE_NL hint. In this statement, the USE_NL 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

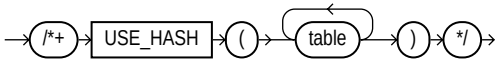
The USE_MERGE hint causes Oracle to join each specified table with another row source with a sort-merge join. The syntax of the USE_MERGE hint is:



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

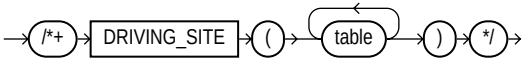
The USE_HASH hint causes Oracle to join each specified table with another row source with a hash join. The syntax of the USE_HASH hint is:



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. 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.

Hints for Parallel Execution

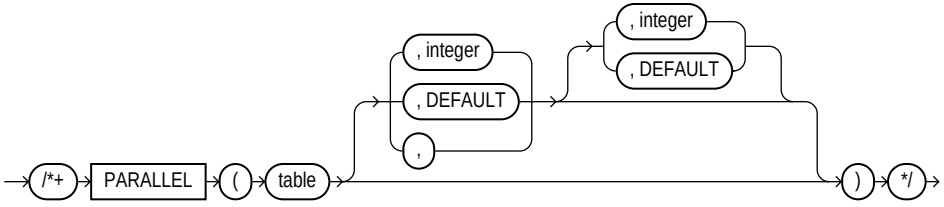
The hints described in this section determine how statements are parallelized or not parallelized when using parallel execution.

- PARALLEL
- NOPARALLEL
- APPEND
- NOAPPEND
- PARALLEL_INDEX
- NOPARALLEL_INDEX

See Also: Chapter 19, "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:

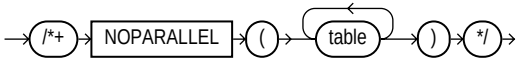
```
SELECT /*+ FULL(scott_emp) PARALLEL(scott_emp, 5) */ ename
  FROM scott.emp scott_emp;
```

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. Note, in general, that hints take precedence over table clauses. Syntax of this hint is:



The following example illustrates the NOPARALLEL hint:

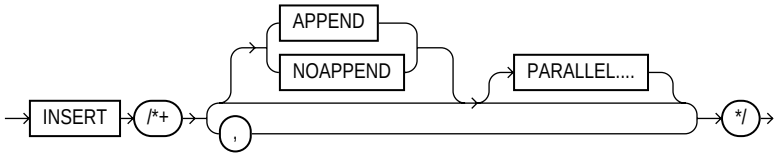
```
SELECT /*+ NOPARALLEL(scott_emp) */ ename
  FROM scott.emp scott_emp;
```

The NOPARALLEL hint is equivalent to specifying the hint

```
/*+ PARALLEL(table,1,1) */
```

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. 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. Note that 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.

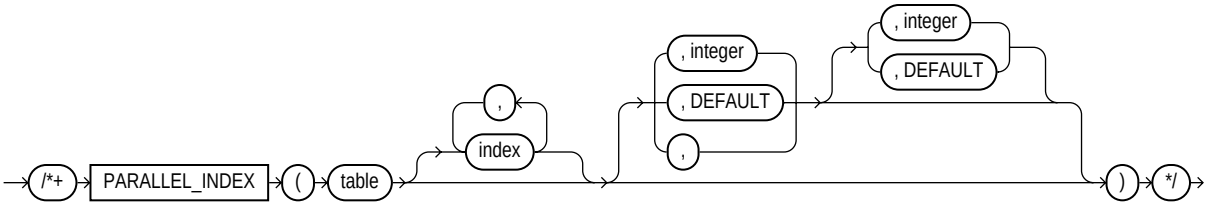
Certain restrictions apply to the APPEND hint; these are detailed in *Oracle8 Concepts*. If any of these restrictions are violated, the hint will be ignored.

NOAPPEND

You can 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.

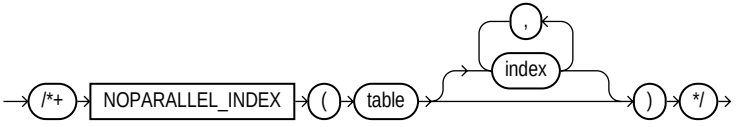
For example:

```
SELECT /*+ PARALLEL_INDEX(table1,index1, 3, 2) +/;
```

In this example there are 3 parallel server processes to be used on each of 2 instances.

NOPARALLEL_INDEX

You can 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:



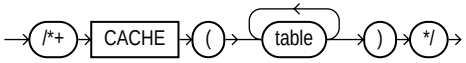
Additional Hints

Several additional hints are included in this section:

- CACHE
- NOCACHE
- MERGE
- NO_MERGE
- PUSH_JOIN_PRED
- NO_PUSH_JOIN_PRED
- PUSH_SUBQ
- STAR_TRANSFORMATION

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. Syntax of this hint is:

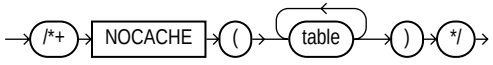


In the following example, the CACHE hint overrides the table's default caching specification:

```
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. Syntax of this hint is:

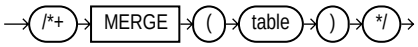


The following example illustrates the NOCACHE hint:

```
SELECT /*+ FULL(scott_emp) NOCACHE(scott_emp) */ ename
  FROM scott.emp scott_emp;
```

MERGE

When the COMPLEX_VIEW_MERGING parameter is set to FALSE, this parameter causes complex views or subqueries to be evaluated before the surrounding query. In this case, you can cause a view to be merged on a per-query basis by using the MERGE hint. Syntax of this hint is:

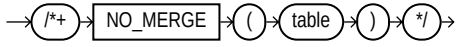


For example:

```
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 COMPLEX_VIEW_MERGING is set to TRUE, you can use the NO_MERGE hint within the view to prevent one particular query from being 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.

PUSH_JOIN_PRED

When the PUSH_JOIN_PREDICATE parameter is TRUE, the optimizer can evaluate, on a cost basis, whether or not to push individual join predicates into the view. This can enable more efficient access paths and join methods, such as transforming hash joins into nested loop joins, and full table scans to index scans.

If the PUSH_JOIN_PREDICATE parameter is FALSE, you can use the PUSH_JOIN_PRED hint to force pushing of a join predicate into the view.

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;
```

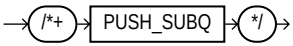
NO_PUSH_JOIN_PRED

If PUSH_JOIN_PREDICATE is TRUE, you can 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. 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.

Note that 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: *Oracle8 Concepts* for a full discussion of star transformation.
Oracle8 Reference describes STAR_TRANSFORMATION_ENABLED; this parameter causes the optimizer to consider performing a star transformation.

Using Hints with Views

Oracle Corporation does not encourage users to use hints inside or on views (or subqueries). This is because views can be defined in one context and used in another; such hints can result in unexpected plans. In particular, hints inside views or on views are handled differently depending on whether or not 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 nonmergeable views, optimization approach and goal hints inside the view are ignored: the top-level query decides the optimization mode.

Since nonmergeable 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 nonmergeable view is similar to a table.

9

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
- Distributed Query Restrictions
- Transparent Gateways
- Summary: Optimizing Performance of Distributed Queries

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:

- Remote Data Dictionary Information
- Remote SQL Statements
- Distributed SQL Statements
- EXPLAIN PLAN and SQL Decomposition
- Partition Views

Remote Data Dictionary Information

If a SQL statement references multiple tables, then the optimizer must determine which columns belong to which tables before it can decompose the SQL statement. For example, with

```
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 the sake of 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 will be 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 8-27 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. In that case 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 8-24
"Hints for Join Operations" on page 8-25

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 23, “The EXPLAIN PLAN Command”

Partition Views

You can utilize partition views to bring together tables that have the same structure, but contain different partitions of data. This can be very useful for a distributed database system, 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 that 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 9-8.

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. Note 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:

```
SQL> r
    1  select lpad(' ',level*3-3)||operation operation,cost,options,
        object_node, other
    2  from plan_table
    3  connect by parent_id = prior id
    4* 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 9-8. 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. Note that 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 dif-

ferent 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 bigger.

- 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. Note that this option is mentioned here only for completeness, not because it is often needed.

10

Data Access Methods

This chapter provides an overview of data access methods that can enhance performance, and warns of situations to avoid. You can use hints to force various approaches. Topics in this chapter include:

- Using Indexes
- Using Bitmap Indexes
- Using Clusters
- Using Hash Clusters

Using Indexes

This section describes:

- When to Create Indexes
- Tuning the Logical Structure
- How to Choose Columns to Index
- How to Choose Composite Indexes
- How to Write Statements that Use Indexes
- How to Write Statements that Avoid Using Indexes
- Assessing the Value of Indexes
- Re-creating an Index
- Using Existing Indexes to Enforce Uniqueness
- Using Enforced 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, you should create indexes on tables that are often 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 can be used for joining to other tables.

This guideline is based on these assumptions:

- Rows with the same value for the column 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 column 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 is excellent at avoiding the use of nonselective indexes within query execution, the SQL engine must continue to maintain all indexes defined against a table whether or not they are ever used. Index maintenance presents 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.

You should drop indexes that are not used. You can detect the indexes that are not referenced in any execution plan by processing all of the application SQL through EXPLAIN PLAN and capturing the resulting plans. Indexes not used in any plan are typically, though not necessarily, nonselective.

Within an application, indexes sometimes have uses that are not immediately apparent from a survey of statement execution plans. In particular, Oracle8 uses "pins" (nontransactional locks) on foreign key indexes to avoid the need for share locks on the parent table when enforcing foreign key constraints. In many applications this foreign key index never (or rarely) supports a query. In the example shown in Figure 10–1, the need to locate all of the order lines for a given product

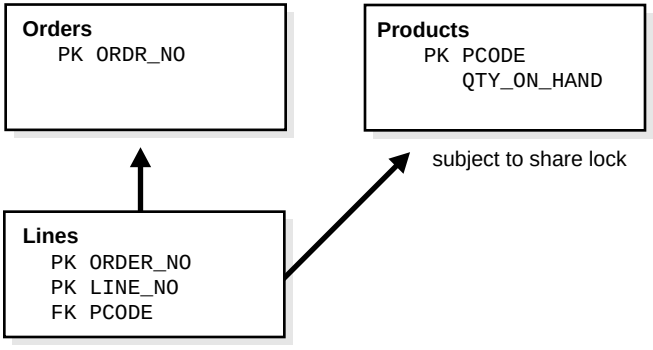
may never arise. However when no index exists with LINES(PCODE) as its leading portion (as described in "How to Choose Composite Indexes" on page 10-6), then Oracle places a share lock on the Products table each time DML is performed against the Lines table. Such a share lock will be a problem only if the Products table itself is subject to frequent DML. In the example shown we might assume that the column QTY_ON_HAND is volatile, and that table level share locks would cause severe contention problems.

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

Fortunately this issue does not normally affect traditional master / detail relationships where the foreign key is generally used as the leading edge of the primary key, as in the example.

Figure 10–1 Foreign Key Constraint



How to Choose Columns to Index

Follow these guidelines for choosing columns to index:

- Consider indexing columns that are used frequently in WHERE clauses.
- Consider indexing columns that are used frequently to join tables in SQL statements. For more information on optimizing joins, see the section "How to Use a Hash Cluster" on page 10-26.
- Only index columns with good selectivity. The selectivity of an index is the percentage of rows in a table having the same value for the indexed column. An index's selectivity is good if few rows have the same value.

  Note: Oracle implicitly creates indexes on the columns of all 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 command. Selectivity calculated in this manner should be interpreted as a percentage.
- Do not use standard B*-tree indexes on columns with few distinct values. Such columns usually have poor selectivity and, therefore, do not optimize performance unless the frequently selected column values appear less frequently than the other column 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 columns 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 column 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. With UPDATE and DELETE, such an index allows Oracle to modify data in the child table without locking the parent table.

- When choosing whether to index a column, consider whether the performance gain for queries is worth the performance loss for INSERT, UPDATE, and DELETE statements and the use of the space required to store the index. You may want to experiment and compare the processing times of your SQL statements with and without indexes. You can measure processing time with the SQL trace facility.

See Also: Chapter 24, "The SQL Trace Facility and TKPROF"
Oracle8 Concepts regarding the effects of foreign keys on locking

How to Choose Composite Indexes

A composite index contains more than one key column. Composite indexes can provide additional advantages over single-column indexes:

better selectivity	Sometimes two or more columns, each with poor selectivity, can be combined to form a composite index with good selectivity.
additional data storage	If all the 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 and Z.

Follow these guidelines for choosing columns for composite indexes:

- Consider creating a composite index on columns 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 column individually.
- If several queries select the same set of columns based on one or more column values, consider creating a composite index containing all of these columns.

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 columns in composite indexes:

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

How to Write 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, you should 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 command 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 means you use to 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.

How to Write 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 or AND_EQUAL hint to force the optimizer to use one index or set of 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 very expensive to maintain. This method, however, is very time and resource expensive. 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.

The parallel query feature utilizes 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.

Fast Full Index Scan

The fast full index scan is an alternative to a full table scan when there is an index that contains all the columns that are needed for the query. 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, no keys can be used, 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 the columns needed from table T2(C2), the optimizer decides to use 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. Note that parallel degree of the index is set independently: the index does *not* inherit the degree of parallelism of the table.
- Make sure that you have analyzed the index, otherwise the optimizer may decide not to use it.

To use this feature you must set the FAST_FULL_SCAN_ENABLED parameter to TRUE.

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 8-22

Re-creating an Index

You may wish to re-create an index in order to compact it and clean up 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 uses the existing index instead of the base table to improve performance.

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. Note that 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 uses the existing index as the basis for the new index. All index storage commands 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 it utilizes the fast full scan feature. It thus 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: *Oracle8 SQL Reference* for more information about the CREATE INDEX and ALTER INDEX commands.

Using Existing Indexes to Enforce Uniqueness

You can use an existing 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 that you rebuild 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. The existing index can be used to enable and enforce the constraint and you can save significant space by not duplicating the index.

Nonunique indexes also have significant advantages when enabling enforced constraints (described in the next section). If you use a nonunique index to enforce a UNIQUE constraint, then when you change the constraint from disabled to enforced, you do not need to rebuild the constraint's index. The existing index is used and the enable operation happens very quickly.

Using Enforced Constraints

An enforced constraint behaves similarly to an enabled constraint. Placing a constraint in the enforced 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 enforced state allows you to enable the constraint without locking the table.

If you change an enforced constraint from disabled to enabled, the table must be locked. No new DML, queries, or DDL can occur, because no mechanism exists to ensure that operations on the table conform to the constraint during the enable operation. The enforced state ensures that no operation violating the constraint can be performed upon the table. Therefore, a constraint can go from enabled to enforced 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, enforced constraints can be enabled in parallel: multiple constraints can be enabled at the same time, and each constraint's validity check can performed using parallel query processors.

The best approach to creating a table with integrity constraints is as follows:

1. For UNIQUE or PRIMARY KEY constraints, create the table with the constraints disabled. For all other constraints, create the table with the constraints enabled.
2. For UNIQUE or PRIMARY KEY constraints, create a nonunique index on the UNIQUE or PRIMARY KEY columns.
3. For UNIQUE or PRIMARY KEY constraints, enable the constraints.
4. Begin entering data into the table.
5. During an IMPORT or load operation, you may wish to disable the constraint for faster processing.
6. Immediately after the IMPORT or load, place the constraint into the ENFORCED state.
7. After enforcing the constraint, perform an ALTER TABLE ... ENABLE CONSTRAINT operation for all of the constraints on the table.

Constraints can be created as enforced. Disabled constraints can be made enforced with the statement

```
ALTER TABLE tablename ENFORCE CONSTRAINT constraint;
```

This statement is about as fast as ALTER TABLE tablename DISABLE, since both statements lock the table but need not check anything.

The IMPORT utility automatically enforces, then enables, named constraints. It enables constraints more slowly if the name is system generated.

See Also: *Oracle8 Concepts* for a complete discussion of integrity constraints.

Using Bitmap Indexes

This section describes:

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

See Also: *Oracle8 Concepts*, for a general introduction to bitmap indexing.

When to Use Bitmap Indexing

This section describes three aspects of the indexing scheme 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.

Storage Considerations

Bitmap indexes can provide considerable storage savings over the use of multicolumn (or concatenated) B*-tree indexes. In databases that contain only B*-tree indexes, a DBA 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 a query that only accesses REGION and GENDER. To completely index the database, the DBA would have to 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. DBAs 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. Although the bitmap indexes may not be quite as efficient during execution as the appropriate composite B*-tree indexes, the space savings more than justifies their use.

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 are not appropriate for OLTP applications with a heavy load of concurrent insert, update, and delete operations. 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 all 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 10-18 only affect the specific operations indicated by the parameter names.

DML and DDL statements such as UPDATE, DELETE, DROP TABLE, and so on, 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 command 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 ROWIDs increase 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.

How to Create a Bitmap Index

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

CREATE BITMAP INDEX ...

All CREATE INDEX parameters except NOSORT are applicable to bitmap indexes. Multi-column (concatenated) bitmap indexes are supported; they can be defined over at most 30 columns. Other SQL statements concerning indexes, such as DROP, ANALYZE, ALTER, and so on, can refer to bitmap indexes without any extra keyword.

Note: The COMPATIBLE initialization parameter must be set to 7.3.2 or higher, for bitmap indexing to be available.

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 indicates to the optimizer the indexes that are cost effective to use. 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.

In deciding which of them actually to use, the optimizer includes nonhinted indexes that look cost effective as well as indexes that are 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 no indexes are named in the hint, all indexes are considered hinted. Hence, the optimizer will try 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, whether or not it considers them cost effective.

See Also: "INDEX_COMBINE" on page 8-21

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.
- In order to make the compressed bitmaps as small as possible, you should 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 23, "The EXPLAIN PLAN Command" for information about bitmap EXPLAIN PLAN output.

Indexing Null Values

Bitmap indexes index null values, 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 !='R';
```

Indexing nulls enables a bitmap minus plan where bitmaps for party equal to 'R' 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
```

Note that 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 impact on performance.

CREATE_BITMAP_AREA_SIZE

This parameter determines the amount of memory allocated for bitmap creation. The default value is 8 Mb. 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. This parameter is not dynamically alterable at the 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. This parameter is not dynamically alterable at the session level.

Using Bitmap Access Plans on Regular B*-tree Indexes

If there exists at least one bitmap index on the table, the optimizer will consider 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.

In order to use bitmap access paths for B*-tree indexes, the ROWIDs stored in the indexes must be converted to bitmaps. Once such a conversion has taken place, 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 in order 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 columns 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 size a bitmap index exactly, you can estimate its size. This section describes how to extrapolate the size of a bitmap index for a table from the computed size of a B*-tree index. It also illustrates how cardinality, NOT NULL constraints and number of distinct values, affects 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 *Oracle8 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 10–2 or Figure 10–3.

For a 1 million row table, Figure 10–2 shows index size on columns with different numbers of distinct values, for B*-tree indexes and bitmap indexes. Using Figure 10–2 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 will vary somewhat from table to table.

Note that 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 10–3 shows similar data for a table with 5 million rows. Note that when cardinality exceeds 100,000, bitmap index size does not increase as fast as it does in Figure 10–2. For a table with more rows, there are more repeating values for a given cardinality.

Figure 10–2 Extrapolating Bitmap Index Size: 1 Million Row Table

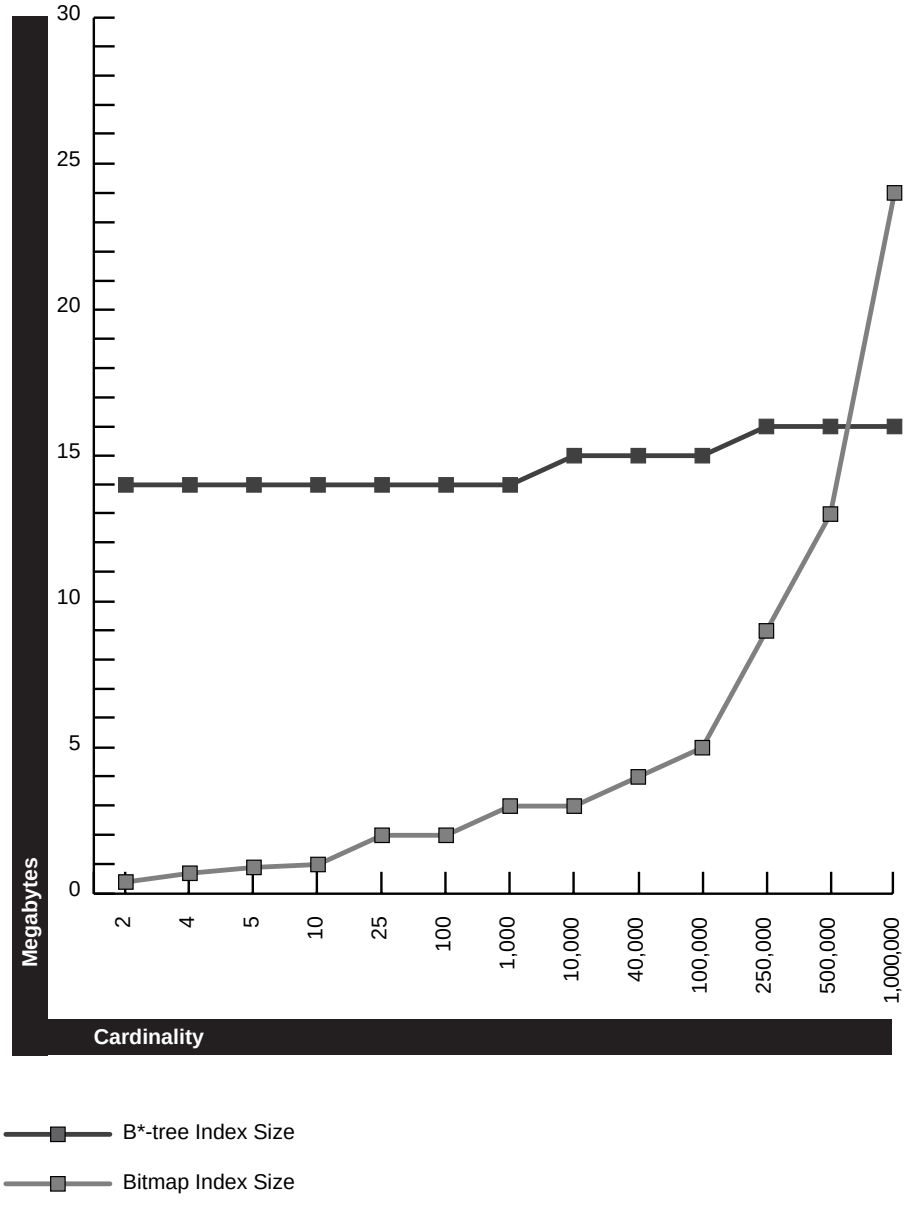
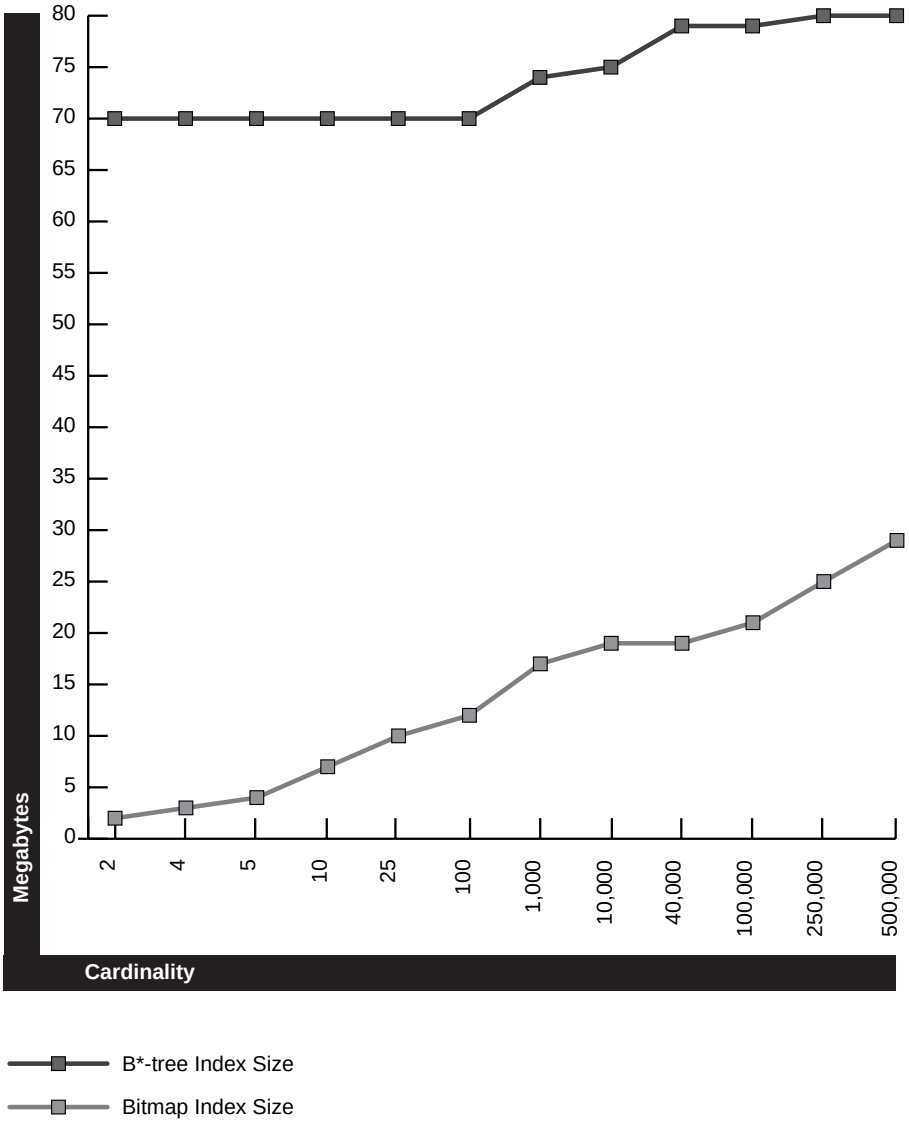


Figure 10–3 Extrapolating Bitmap Index Size: 5 Million Row Table



Bitmap Index Restrictions

Bitmap indexes have the following restrictions:

- For bitmap indexes with direct load, the SORTED_INDEX flag does not apply.
- Performing an ALTER TABLE command that adds or modifies a bitmap-indexed column may cause indexes to be invalidated.
- Bitmap indexes are not considered by the rule-based optimizer.
- Bitmap indexes cannot be used for referential integrity checking.

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 command.

See Also: For more information on creating clusters, see the *Oracle8 Application Developer's Guide*.

Using Hash Clusters

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

- When to Use a Hash Cluster
- How to Use a Hash Cluster

When to Use a Hash Cluster

Follow these guidelines for choosing when to use hash clusters:

- Consider using hash clusters to store tables that are 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.

How to Use a Hash Cluster

To create a hash cluster, use the CREATE CLUSTER command with the HASH and HASHKEYS parameters.

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 the *Oracle8 Application Developer's Guide*.

11

Oracle8 Transaction Modes

This chapter describes the different modes in which read consistency is performed. Topics in this chapter include:

- Using Discrete Transactions
- Using Serializable Transactions

Using Discrete Transactions

You can improve the performance of short, nondistributed transactions by using the BEGIN_DISCRETE_TRANSACTION procedure. This procedure streamlines transaction processing so short transactions can execute more rapidly. This section describes:

- Deciding When to Use Discrete Transactions
- How Discrete Transactions Work
- Errors During Discrete Transactions
- Usage Notes
- Example

Deciding When to Use Discrete Transactions

Discrete transaction processing is useful for transactions that:

- modify only a few database blocks
- never change an individual database block more than once per transaction
- do not modify data likely to be requested by long-running queries
- do not need to see the new value of data after modifying the data
- do not modify tables containing any LONG values

In deciding to use discrete transactions, you should consider the following factors:

- Can the transaction be designed to work within the constraints placed on discrete transactions, as described in "Usage Notes" on page 11-4?
- 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. Although discrete transactions can be used only for a subset of all transactions, for sophisticated users with advanced application requirements, where speed is the most critical factor, the performance improvements can make working within the design constraints worthwhile.

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. This eliminates the 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.

To use the BEGIN_DISCRETE_TRANSACTION procedure, the DISCRETE_TRANSACTIONS_ENABLED initialization parameter must be set to TRUE. If this parameter is set to FALSE, all calls to BEGIN_DISCRETE_TRANSACTION are ignored and transactions requesting this service are handled as standard transactions.

See Also: *Oracle8 Reference* for information about setting initialization parameters.

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;
```

Note 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: *Oracle8 SQL Reference* for more information on the syntax of SET TRANSACTION and ALTER SESSION.

12

Managing SQL and Shared PL/SQL Areas

This chapter explains the use of shared SQL to improve performance. Topics in this chapter include

- Introduction
- Comparing SQL Statements and PL/SQL Blocks
- Keeping Shared SQL and PL/SQL in the Shared Pool

Introduction

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 are aged out of the shared pool by way of a least recently used 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.

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 quite 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 Being Aged 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, an enormous amount of maintenance must be done 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 Oracle8 uses segmented codes to reduce the need for large areas of contiguous memory, it may still be valuable for performance reasons for you to pin large objects in memory.

Preventing Objects from Being Aged Out

The DBMS_SHARED_POOL package lets you keep objects in shared memory, so that they will not be aged 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 early these SQL and PL/SQL areas (before memory fragmentation occurs), the objects can be kept in memory, instead of aging out with the normal LRU mechanism. This procedure ensures that memory is available and prevents sudden, seemingly 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, users' response time is affected because of the large number of smaller objects that need to be aged out from the shared pool to make room (due to memory fragmentation). In some cases, there may be insufficient memory to load the large objects.

DBMS_SHARED_POOL is also useful for frequently executed triggers. You may want to keep compiled triggers on frequently used tables in the shared pool.

Note in addition that DBMS_SHARED_POOL supports sequences. Sequence numbers are lost when a sequence is aged 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/cursors you would like pinned 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, this procedure prevents the memory fragmentation that could result from pinning a large chunk 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 Server Manager or 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 Server Manager or SQL*Plus commands:

```
SQL> SET SERVEROUTPUT ON SIZE 2000
SQL> EXECUTE DBMS_SHARED_POOL.SIZES(2000);
```

DBMS_SHARED_POOL.KEEP

This procedure lets you keep an object in the shared pool. This procedure may not be supported in future releases.

```
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	procedure
C	cursor
R	trigger
Q	sequence

DBMS_SHARED_POOL.UNKEEP

This procedure allows an object that you have requested to be kept in the shared pool now to be aged out of the shared pool.

Note: This procedure may not be supported in the future.

```
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	procedure
C	cursor
R	trigger
Q	sequence

Part IV

Optimizing Oracle Instance Performances

Part IV describes how to tune various elements of your database system in order to optimize performance of an Oracle instance. The chapters are:

- Chapter 13, “Tuning CPU Resources”
- Chapter 14, “Tuning Memory Allocation”
- Chapter 15, “Tuning I/O”
- Chapter 16, “Tuning Networks”
- Chapter 17, “Tuning the Operating System”
- Chapter 18, “Tuning Resource Contention”

13

Tuning CPU Resources

This chapter describes how to identify and solve problems with central processing unit (CPU) resources. Topics in this chapter include

- Understanding CPU Problems
- How to Detect and Solve CPU Problems
- Solving CPU Problems by Changing System Architecture

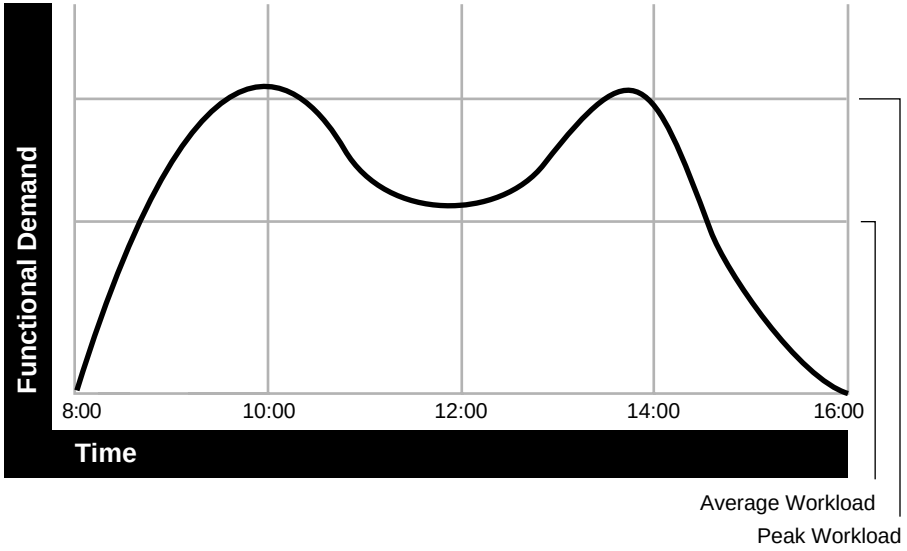
Understanding CPU Problems

Establish appropriate expectations for the amount of CPU resources your system should be using. You can then distinguish whether or not sufficient CPU resources are available, and know when your system is consuming too much of those resources. Begin by determining the amount of CPU resources the Oracle instance utilizes in three cases:

- while the machine is idle
- at average workload
- at peak workload

Workload is a very 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 understandable and 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 peak workload. For example, Figure 13–1 illustrates workload over time for an application which has peak periods at 10:00 AM and 2:00 PM.

Figure 13–1 Average Workload and Peak Workload



This example application has 100 users working 8 hours a day, for a total of 800 hours per day. If each user enters one transaction every 5 minutes, this would mean

9,600 transactions per day. Over the course of 8 hours, 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 form peaks and valleys—and in this context 20 transactions per minute can be understood as merely a minimum requirement. If the peak rate you need to achieve is 120 transactions per minute, then you must configure a system that can support this peak workload.

For this example, assume that at peak workload Oracle can use 90% of the CPU resource. For a period of average workload, then, Oracle should be using no more than about 15% of the available CPU resource.

15% = 20 tpm / 120 tpm * 90%

If the system requires 50% of the CPU resource to achieve 20 transactions per minute, then it is clear that a problem exists: the system cannot possibly achieve 120 transactions per minute using 90% of the CPU. However, if you could tune this system so that it does achieve 20 transactions per minute using only 15% of the CPU, then (assuming linear scalability) the system might indeed attain 120 transactions per minute using 90% of the CPU resources.

Note that as users are added to an application over time, the average 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 before.

How to Detect and Solve CPU Problems

If you suspect a problem with CPU usage, you must evaluate two areas:

- Checking System CPU Utilization
- Checking Oracle CPU Utilization

Checking System CPU Utilization

Oracle statistics report CPU utilization only of Oracle sessions, whereas every process running on your system affects the available CPU resources. Effort spent tuning non-Oracle factors can thus result in better Oracle performance.

Use operating system monitoring tools to see 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.

Performance Monitor is used on NT systems to examine CPU utilization. It 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.)

Attention: This section describes how to check system CPU utilization on most UNIX-based and NT systems. For other platforms, please check your operating system documentation.

Memory Management

Check the following memory management issues:

Paging and Swapping. Use the appropriate tools (such as **sar** or **vmstat** on UNIX or Performance Monitor on NT) to investigate the cause of paging and swapping, should they occur.

Oversize Page Tables. On UNIX systems, if the processing space becomes too large, it may result in the page tables becoming too large. This is not an issue on NT systems.

I/O Management

Check the following I/O management issues:

Thrashing. Make sure that your workloads fits in memory so that the machine is not thrashing (swapping and paging processes in and out of memory). The operating system allocates fixed slices of time during which CPU resources are available to your process. If the process squanders a large portion of each time slice checking to be sure that it can run, that all needed components are in the machine, it 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. This results in a lot of overhead that must be completed before the message is actually sent. To alleviate this problem you can 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.

Process Management

Check the following process management issues:

Scheduling and Switching. The operating system may spend a lot of time in scheduling and switching processes. Examine the way in which you are using the operating system: you could be using too many processes. On NT systems, do not overload your server with a great deal of non-Oracle processes.

Context Switching. Due to operating system specific characteristics, your system could be spending a lot of time in context switches. This could be expensive, especially with a very large SGA. Note that 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 on the fly; then they exit the process, and create a new one so that the process is re-created and destroyed all the time. This is very CPU intensive, especially with large SGAs, because you have to build up the page tables each time. The problem is aggravated when you nail or lock shared memory, because you have to touch 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 query and multithreaded server are areas of concern here if MINSERVICE has been set too low (set to 10, for example, when you need 20).

For the user, doing small lookups may not be wise. In a situation like this, it becomes inefficient for the user and for the system as well.

Checking Oracle CPU Utilization

This section explains how to examine the processes running in Oracle. Two dynamic performance views provide information on Oracle processes:

- V$SYSSTAT shows Oracle CPU usage for all sessions. The statistic “CPU used by this session” actually shows the aggregate CPU used by all sessions.
- V$SESSTAT shows Oracle CPU usage per session. You can use this view to see 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-based systems 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 utilized 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. Go back to the system and find out what process is using up the CPU time. Identify the process, determine why it is using so much CPU time, and see if you can tune it.

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

Ineffective SQL sharing can result in reparsing.

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)');
```

   In interpreting these statistics, remember

   - 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 you will spend waiting. Note that

   - wait time / parse count = average wait time per parse
   - The average wait time should be extremely low, approaching zero. (V$SYSSTAT also indicates the average wait time per parse.)

2. Next, query V$SQLAREA to find frequently reparsed statements:

```
SELECT SQL_TEXT, PARSE_CALLS, EXECUTIONS FROM V$SQLAREA
ORDER BY PARSE_CALLS;
```

3. Now that you have identified problematic statements, you have the following three options for tuning them:

   - Rewrite the application so statements do not continually reparse.
   - If this is not possible, 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. Change to bind variables in order to reduce parsing.

Inefficient SQL Statements

Inefficient SQL statements can consume large amounts of CPU resource. 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 7-6

Read Consistency

Your system could spend a lot of time rolling back changes to blocks in order to maintain a consistent view.

- 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 has to 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:

$$\frac{1 + \textit{value of "free buffers inspected"}}{\textit{"free buffers inspected"}} = \textit{avg. buffers scanned}$$

  Normally you would expect to see 1 or 2 buffers scanned, on average. If more than this number are being scanned, increase the size of the buffer cache or tune the DBW*n* process(es).

You can apply 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 buffer cache size or tune DBW*n*.

Scalability Limitations Within the Application

In most of this CPU tuning discussion we assume linear scalability, but this is never actually the case. How flat or nonlinear the scalability is indicates how far away from the ideal you are. Problems in your application might be hurting scalability: examples include too many indexes, right-hand index problems, too much data in blocks, or not partitioning the data. Contention problems like these waste CPU cycles and prevent the application from attaining linear scalability.

Latch Contention

Latch contention is a symptom; it is not normally the cause of CPU problems. Your task is to translate the latch contention to an application area: track down the contention to determine which part of your application is poorly written.

The spin count may be set too high. Some other process may be holding a latch that your process is attempting to get, and your process may be spinning and spinning in an effort to get the latch. After a while your process may go to sleep before waking up to repeat its ineffectual spinning.

- Check the Oracle latch statistics. The “latch free” event in V$SYSTEM_EVENT shows how long you have been waiting on latches. If there is no latch contention, this statistic will not appear.
- Check the value of the SPINCOUNT initialization parameter. SPINCOUNT depends heavily on CPU speed: appropriate values vary significantly from platform to platform.

  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 up a great deal of CPU time by actively spinning and waiting.
- Look for the ratio of CPUs to processes. If there are large numbers of both, then many process can run. But if a single process is holding a latch on a system with ten CPUs, and that process should be rescheduled so that it is not run-

ning, then ten other processes may run ineffectively, trying to get the same latch. That situation will waste, in parallel, some CPU resource.

- Check V$LATCH_MISSES, which indicates where in the Oracle code most of the contention occurs.

Solving CPU Problems by Changing System Architecture

If you have reached the limit of CPU power available on your system, and have exhausted all means of tuning its CPU usage, then you must consider redesigning your system. Consider whether moving to a different architecture might result in adequate CPU power. This section describes various possibilities.

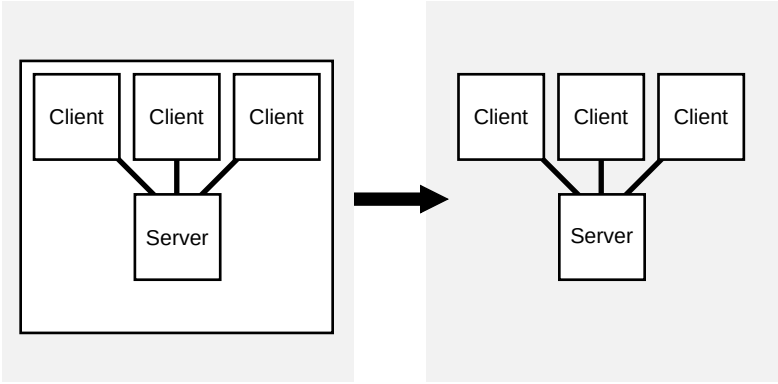
- Single Tier to Two-Tier
- Multi-Tier: Using Smaller Client Machines
- Two-Tier to Three-Tier: Using a Transaction Processing Monitor
- Three-Tier: Using Multiple TP Monitors
- Oracle Parallel Server

Attention: If you are running a multi-tier system, check all levels for CPU utilization. For example, on a three-tier system you might learn that your server is mostly idle and your second tier is completely busy. The solution then would be clear: tune the second tier, rather than the server or the third tier. In a multi-tier situation, 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 clients and server all running on a single machine (single tier) to a two-tier client/server configuration could help to relieve CPU problems.

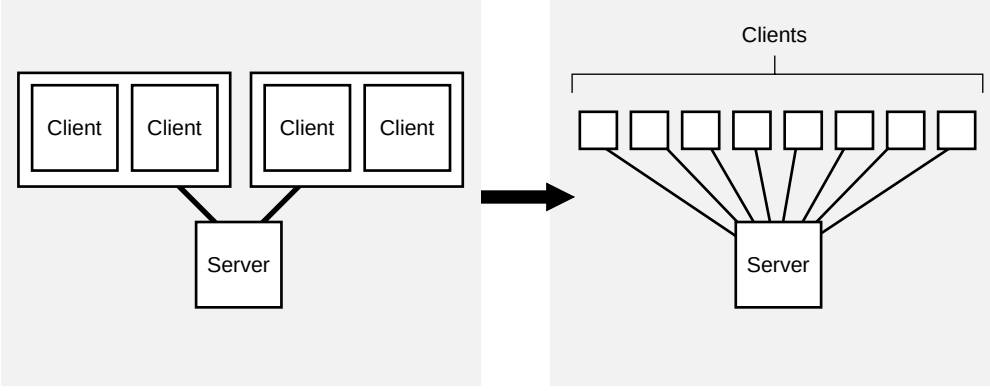
Figure 13–2 Single Tier to Two-Tier



Multi-Tier: Using Smaller Client Machines

Consider whether CPU usage might be improved if you used smaller clients, rather than multiple clients on bigger machines. This strategy may be helpful with either two-tier or three-tier configurations.

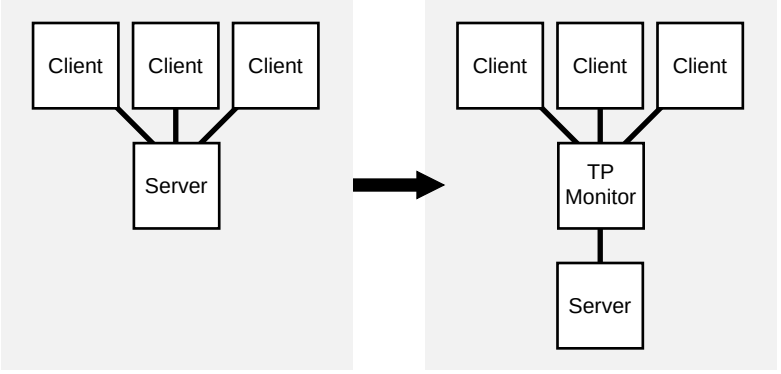
Figure 13–3 Multi-Tier Using Smaller Clients



Two-Tier to Three-Tier: Using a Transaction Processing Monitor

If your system currently runs with multiple layers, consider whether moving from a two-tier to three-tier configuration, introducing the use of a transaction processing monitor, might be a good solution.

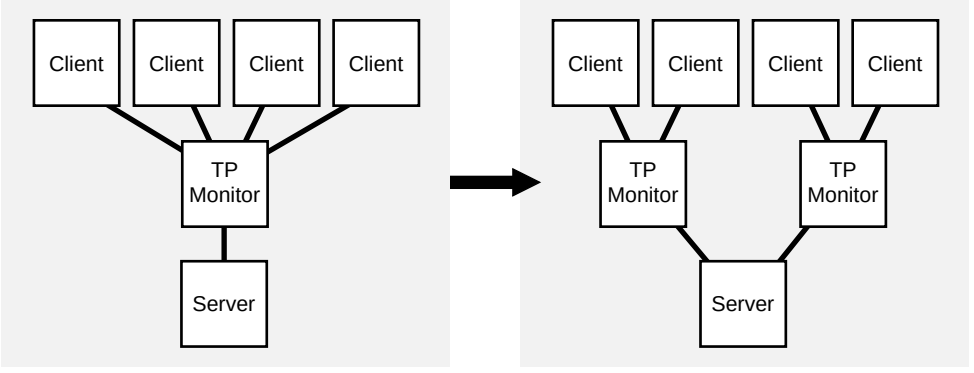
Figure 13–4 Two-Tier to Three-Tier



Three-Tier: Using Multiple TP Monitors

Consider whether using multiple transaction processing monitors might be a good solution.

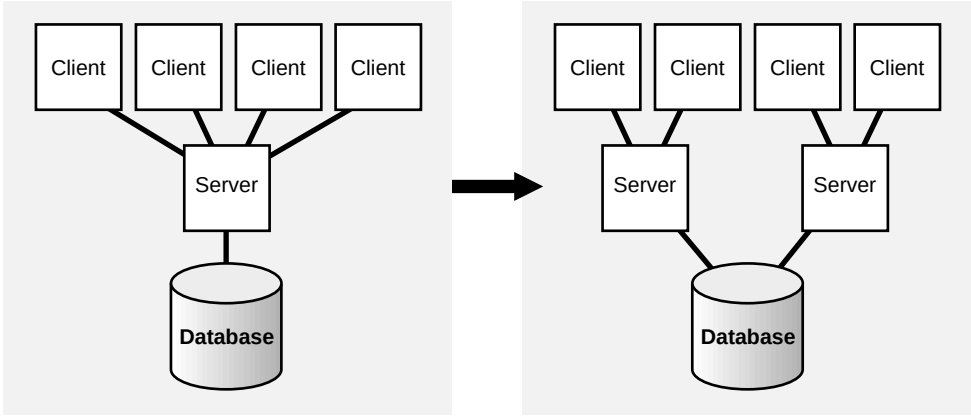
Figure 13–5 Three-Tier with Multiple TP Monitors



Oracle Parallel Server

Consider whether your CPU problems could be solved by incorporating Oracle Parallel Server.

Figure 13–6 Oracle Parallel Server



14

Tuning Memory Allocation

This chapter explains how to allocate memory to database structures. Proper sizing of these structures can greatly improve database performance. The following topics are covered:

- Understanding Memory Allocation Issues
- How to Detect Memory Allocation Problems
- How to Solve Memory Allocation Problems
  - Tuning Operating System Memory Requirements
  - Tuning the Redo Log Buffer
  - Tuning Private SQL and PL/SQL Areas
  - Tuning the Shared Pool
  - Tuning the Buffer Cache
  - Tuning Multiple Buffer Pools
  - Tuning Sort Areas
  - Reallocating Memory
  - Reducing Total Memory Usage

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, you should 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.

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 disk storage. A segment is stored in many blocks.
buffer	A container in memory for a block. At any point in time a buffer holds a single block. Over time a buffer may hold different blocks, as when a new block is needed an old block is discarded and replaced with the 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 that have been allocated for a specific type of database object such as a table, index, cluster.

See Also: Chapter 15, "Tuning I/O", shows you how to perform I/O as efficiently as possible.

How to Detect Memory Allocation Problems

When you use operating system tools such as **ps -efl** or **ps - aux** on UNIX-based systems to look at the size of Oracle processes, you may notice that the processes seem relatively large. To interpret the statistics shown, you must 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 4K), 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.

How to Solve Memory Allocation Problems

The rest of this chapter explains in detail how to tune memory allocation. For best results, you should tackle memory issues in the order they are 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 Enough 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 any of these places:

- real memory
- virtual memory
- expanded storage
- disk

The operating system may also move information from one storage location to another, a process 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 you have allocated.

See Also: "Oversubscribe, with Attention to Paging" on page 19-39

Fitting the System Global Area into Main Memory

Since the purpose of the System Global Area (SGA) is to store data in memory for fast access, the SGA should always be contained in main memory. If pages of the SGA are swapped to disk, its data is no longer so 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 will always be in memory because they are always being referenced. Some of the cold parts may be paged out, and a performance penalty may result from bringing them back in. A performance problem is very likely, however, if the hot part of the SGA cannot stay in memory.

Remember that 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 prebuilt 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 that 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 also increase the amount of time needed for process startup, because every process that starts must attach 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 is started. This approach may be useful with some applications, but not with all applications. Overhead may be significant if your system creates and destroys processes all the time (by doing continual logon / logoff, for example).

The advantage that PRE_PAGE_SGA can afford depends on page size. For example, if the SGA is 80 MB in size, and the page size is 4K, then 20,000 pages must be touched in order 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). Note that 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 Server Manager statement:

```
SVRMGR> SHOW SGA
```

The output of this statement might look like this:

```
Total System Global Area      3554188 bytes
              Fixed Size        22208 bytes
           Variable Size      3376332 bytes
         Database Buffers       122880 bytes
             Redo Buffers        32768 bytes
```

Some operating systems for IBM mainframe computers are equipped with 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 stays in main memory. If your operating system has expanded storage, you can take advantage of it by allocating a larger SGA despite the resulting paging.

Allocating Enough 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 in order to use their application with Oracle.

Depending on your operating system, these resources may include:

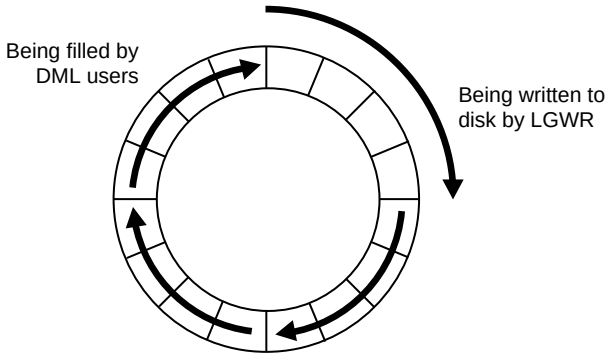
- the Oracle executable image
- the SGA
- Oracle application tools
- application-specific data

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, which is fixed in size. On machines with fast processors and relatively slow disks, the processor(s) may be filling the rest of the buffer in the time it takes the redo log writer to move a portion of the buffer out to disk. The log writer process (LGWR) is always started when the buffer begins to fill. For this reason a larger buffer makes it less likely that new entries will collide with the part of the buffer still being written.

Figure 14–1 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, then 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. If a lot of reparsing occurs, less memory is needed. If you reduce reparsing (by creating more SQL statements), then the memory requirement on the client side increases. 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 that are satisfied by access to the library cache as well as parse calls that result 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 14-13.

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_count, executions
FROM V$SQLAREA
```

When the parse_count value is close to the execution value for a given statement, you may be continually reparsing that particular SQL statement.

Technique 3

You can also identify unnecessary parse calls by identifying the session that gives rise to them. It may be that particular batch programs or types of application do most of the reparsing. Execute the following query:

```
SELECT * FROM V$STATNAME
WHERE name in ('parse_count (hard)','execute_count')
```

The results of the query will look something like this:

```
statistic#,   name
------------  ---------
100           parse_count
90            execute_count
```

Then run a query like the following:

```
SELECT * FROM V$SESSTAT
WHERE statistics# in (90,100)
ORDER BY value, sid;
```

The result will be 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 is reused for a new SQL statement.

Be sure that your application can open enough private SQL areas to accommodate all of 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. Note that 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 Shared Pool with the Multithreaded Server
- 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 14-25.

The algorithm that Oracle uses to manage data in the shared pool tends to hold dictionary 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 the multithreaded server architecture.

In the shared pool, some of the caches are dynamic—they grow or shrink as needed. These dynamic caches include the library cache and the data dictionary cache. Objects are paged out of these caches if in the shared pool runs out of room. For this reason you may have to increase shared pool size if the “hot” (often needed) 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, you should allocate sufficient memory for the shared pool first.

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-side programs.

If the shared pool is too small, then the server must dedicate resources to managing the limited space available. This consumes CPU resources and causes contention, because restrictions must be imposed 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 over a specific 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.)

```
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. The instantaneous value can be reported 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 will have little or no beneficial effect. However, just because the shared pool is full does not necessarily mean that there is a problem. If the ratios discussed above are close to 1, there is no need to increase the pool size.

Once an entry has been loaded into the shared pool it cannot be moved. As more entries are loaded, the areas of free memory are broken up and the shared pool may become fragmented. On UNIX-based systems, 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.

Oracle8 uses segmented codes to reduce the need for large areas of contiguous memory. For performance reasons, however, you may still want to pin a large object in memory. Using the DBMS_SHARED_POOL package, you can keep large objects permanently pinned in the shared pool.

The library cache hit ratio and row cache hit ratio are important. If free memory is close to zero and either the library cache hit ratio or the row cache hit ratio is less than 0.95, then increase the shared pool until the ratios stop improving.

Tuning the Library Cache

The library cache contains shared SQL and PL/SQL areas. This section tells you how to tune the library cache. Information presented here about shared SQL areas and SQL statements also applies to shared PL/SQL areas and PL/SQL blocks.

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.

Determine whether misses on the library cache are affecting the performance of Oracle by querying the dynamic performance table V$LIBRARYCACHE.

The V$LIBRARYCACHE Table You can monitor statistics reflecting library cache activity by examining the dynamic performance table V$LIBRARYCACHE. These statistics reflect all library cache activity since the most recent instance startup. 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 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 had 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 for 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 14-25

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 8-6.

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 speed 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.

- If the value of this parameter is 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 the value of this parameter is 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. A least recently used (LRU) algorithm ages out 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 Shared Pool with the Multithreaded Server

In the multithreaded server architecture, Oracle stores session information in the shared pool rather than in the memory of user processes. Session information includes private SQL areas. If you are using the multithreaded server architecture, you may need to make your shared pool larger to accommodate session information. You can increase the size of the shared pool by increasing the value of the SHARED_POOL_SIZE initialization parameter. This section discusses measuring the size of session information by querying the dynamic performance table V$SESSTAT.

With very high numbers of connected users, the only way to reduce memory usage to an acceptable level may be to go to three-tier connections. This by-product of using a TP monitor is feasible only with a pure transactional model, because no locks or uncommitted DML can be held between calls. Oracle's multithreaded server (MTS) is much less restrictive of the application design than a TP monitor. It dramatically reduces operating system process count, because it normally requires only 5 threads per CPU. It still requires a minimum of about 300K bytes of context per connected user.

The V$SESSTAT Table

Oracle collects statistics on total memory used by a session and stores them in the dynamic performance table V$SESSTAT. By default, this table 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 14-8.

Querying the V$SESSTAT Table

You can use this query to decide how much larger to make the shared pool if you are using a multithreaded 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 Table

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 multithreaded 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 two initialization parameters:

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 on the system.
SHARED_POOL_RESERVED_MIN_ALLOC	Controls allocation for the reserved memory. To create a reserved list, SHARED_POOL_RESERVED_SIZE must be greater than SHARED_POOL_RESERVED_MIN_ALLOC. Only allocations larger than SHARED_POOL_RESERVED_POOL_MIN_ALLOC can allocate space from the reserved list if a chunk of memory of sufficient size is not found on the shared pool's free lists. The default value of SHARED_POOL_RESERVED_MIN_ALLOC should be adequate for most systems.

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 the DBA 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 should be sufficient, if you have already done some tuning of the shared pool. The default value for SHARED_POOL_RESERVED_MIN_ALLOC is usually adequate. If you increase this value, then the database will allow fewer allocations from the reserved list and will request more memory from the shared pool list.

Ideally, you should make 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 SGA, the goal is to have REQUEST_MISSES = 0. If the system is constrained for OS memory, the goal is as follows:

- REQUEST_FAILURES = 0 or not increasing
- LAST_FAILURE_SIZE > SHARED_POOL_RESERVED_MIN_ALLOC
- AVG_FREE_SIZE > SHARED_POOL_RESERVED_MIN_ALLOC

If neither the second nor the third of these goals is met, increase SHARED_POOL_RESERVED_SIZE. Also increase SHARED_POOL_SIZE by the same amount, because the reserved list is taken from the shared pool.

See Also: *Oracle8 Reference* for details on setting the LARGE_POOL_SIZE and LARGE_POOL_MIN_ALLOC parameters

SHARED_POOL_ RESERVED_SIZE Too Small

The reserved pool is too small when:

- REQUEST_FAILURES > 0 (and increasing)

and at least one of the following is true:

- LAST_FAILURE_SIZE > SHARED_POOL_RESERVED_MIN_ALLOC
- MAX_FREE_SIZE < SHARED_POOL_RESERVED_MIN_ALLOC
- FREE_SPACE < SHARED_POOL_RESERVED_MIN_ALLOC

You have two options, depending on SGA size constraints:

- Increase SHARED_POOL_RESERVED_SIZE and SHARED_POOL_SIZE accordingly.
- Increase SHARED_POOL_RESERVED_MIN_ALLOC (but you may need to increase SHARED_POOL_SIZE).

The first option increases the amount of memory available on the reserved list without having an impact on users not allocating memory from the reserved list. The second options reduces the number of allocations allowed to use memory from the reserved list; doing so, however, increases the normal shared pool, which may have an impact 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

You have two options:

- Decrease SHARED_POOL_RESERVED_SIZE.
- Decrease SHARED_POOL_RESERVED_MIN_ALLOC (if not the default value).

SHARED_POOL_SIZE Too Small

The V$SHARED_POOL_RESERVED fixed table can also indicate when SHARED_POOL_SIZE is too small. This may be the case if:

- REQUEST_FAILURES > 0 and increasing
- LAST_FAILURE_SIZE < SHARED_POOL_RESERVED_MIN_ALLOC

Then you have two options, if you have enabled the reserved list:

- Decrease SHARED_POOL_RESERVED_SIZE.
- Decrease SHARED_POOL_RESERVED_MIN_ALLOC (if set larger than the default).

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. Note that Oracle bypasses the buffer cache for sorting and parallel reads. For operations that do use the buffer cache, this section explains:

- Evaluating Buffer Cache Activity by Means of the Cache Hit Ratio
- Raising 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 significant time (typically in excess of 15 msec) and also increases the CPU resource 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 table V$SYSSTAT. By default, this table is available only to the user SYS and to users (such as SYSTEM) who have been granted SELECT ANY TABLE system privilege. Information in the V$SYSSTAT table 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	The value of 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:

Hit Ratio = 1 - (*physical reads* / (*db block gets* + *consistent gets*))

Based on the statistics obtained by the example query, the buffer cache hit ratio is 94%.

Evaluating the Cache Hit Ratio

When looking at the cache hit ratio, bear in mind that blocks encountered during a "long" full table scan are not put at the head of the LRU list; therefore repeated scanning will not cause the blocks to be cached.

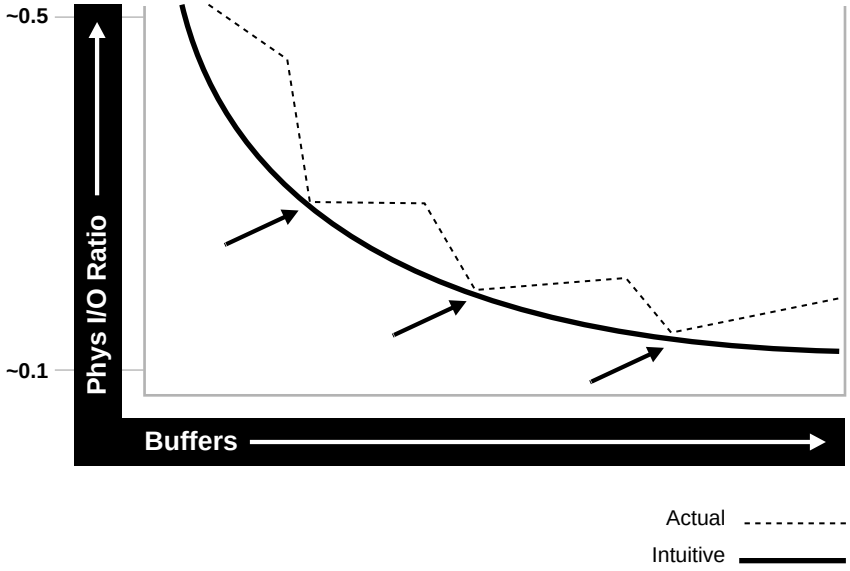
Repeated scanning of the same large table is rarely the most efficient approach. It may be better to perform all of the processing in a single pass, even if this means that the overnight batch suite can no longer be implemented as a SQL*Plus script which contains no PL/SQL. The solution therefore lies at the design or implementation level.

Note: The CACHE_SIZE_THRESHOLD parameter sets the maximum size of a table to be cached, in blocks; it is equal to one tenth of DB_BLOCK_BUFFERS. On a per-table basis, this parameter enables you to determine which tables should and should not be cached.

Production sites running with thousands or tens of thousands of buffers rarely use memory effectively. In any large database running an OLTP application, in any given unit of time, most rows will be 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 cache hit ratio and 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 the following figure, only narrow bands of values of DB_BLOCK_BUFFERS are worth considering. The effect is not completely intuitive.

Figure 14–2 Buffer Pool Cache Hit Ratio



Attention: A common mistake is to continue increasing the value of DB_BLOCK_BUFFERS. Such increases will make no difference at all if you are doing full table scans and other operations that do not even use the buffer pool.

As a rule of thumb, increase DB_BLOCK_BUFFERS while:

- 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 the instance is always started in exclusive mode.

Perform a query like the following:

```
SELECT file#, COUNT(block#), COUNT (DISTINCT file# || block#)
FROM V$BH
GROUP BY file#
```

Raising Cache Hit Ratio by Reducing Buffer Cache Misses

If your hit ratio is low, perhaps 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.

The V$RECENT_BUCKET View

The virtual table V$RECENT_BUCKET contains statistics that estimate the performance of a larger cache. Each row in the table reflects the relative performance value of adding a buffer to the cache. This table can only be accessed only by the user SYS. The following are the columns of the V$RECENT_BUCKET view

ROWNUM — The value of this column is one less than the number of buffers that would potentially be added to the cache.

COUNT — The value of this column is the number of additional cache hits that would be obtained by adding buffer number ROWNUM+1 to the cache.

For example, in the first row of the table, the ROWNUM value is 0 and the COUNT value is the number of cache hits to be gained by adding the first additional buffer to the cache. In the second row, the ROWNUM value is 1 and the COUNT value is the number of cache hits for the second additional buffer.

Note: The GV$CURRENT_BUCKET and GV$RECENT_BUCKET views provide the instance identifier (INST_ID) along with the count. If necessary, you can infer

the INDX value: the *n*th entry for a particular instance would reflect index *n* for that instance.

Enabling the V$RECENT_BUCKET View

The collection of statistics in the V$RECENT_BUCKET view is controlled by the initialization parameter DB_BLOCK_LRU_EXTENDED_STATISTICS. The value of this parameter determines the number of rows in the V$RECENT_BUCKET view. The default value of this parameter is 0, which means the default behavior is not to collect statistics.

To enable the collection of statistics in the V$RECENT_BUCKET view, set the value of DB_BLOCK_LRU_EXTENDED_STATISTICS. For example, if you set the value of the parameter to 100, Oracle will collect 100 rows of statistics, each row reflecting the addition of one buffer, up to 100 extra buffers.

Collecting these statistics incurs some performance overhead, which is proportional to the number of rows in the table. To avoid this overhead, collect statistics only when you are tuning the buffer cache; disable the collection of statistics when you are finished tuning.

Querying the V$RECENT_BUCKET View

From the information in the V$RECENT_BUCKET view, you can predict the potential gains of increasing the cache size. For example, to determine how many more cache hits would occur if you added 20 buffers to the cache, query the V$RECENT_BUCKET view with the following SQL statement:

```
SELECT SUM(count) ach
   FROM V$RECENT_BUCKET
   WHERE ROWNUM < 20;
```

You can also determine how these additional cache hits would affect the hit ratio. Use the following formula to calculate the hit ratio based on the values of the statistics *db block gets*, *consistent gets*, and *physical reads* and the number of additional cache hits (ACH) returned by the query:

Hit Ratio = 1 - (*physical reads* - ACH / (*db block gets* + *consistent gets*))

Grouping Rows in the V$RECENT_BUCKET View

Another way to examine the V$RECENT_BUCKET view is to group the additional buffers in large intervals. You can query the table with a SQL statement similar to this:

```
SELECT 250*TRUNC(ROWNUM/250)+1||' to '||250*(TRUNC(ROWNUM/250)+1)
"Interval", SUM(count) "Buffer Cache Hits"
   FROM V$RECENT_BUCKET
   GROUP BY TRUNC(ROWNUM/250);
```

The result of this query might look like this:

```
Interval          Buffer Cache Hits
--------------- --------------------
1 to 250                     16080
251 to 500                   10950
501 to 750                     710
751 to 1000                  23140
```

where:

Interval	Is the interval of additional buffers to be added to the cache.
Buffer Cache Hits	Is the number of additional cache hits to be gained by adding the buffers in the INTERVAL column.

Examining the query output leads to these observations:

- If 250 buffers were added to the cache, 16,080 cache hits would be gained.
- If 250 more buffers were added for a total of 500 additional buffers, 10,950 cache hits would be gained in addition to the 16,080 cache hits from the first 250 buffers. This means that adding 500 buffers would yield a total of 27,030 additional cache hits.
- If 250 more buffers were added for a total of 750 additional buffers, 710 cache hits would be gained, yielding a total of 27,740 additional cache hits.
- If 250 buffers were added to the cache for a total of 1000 additional buffers, 23,140 cache hits would be gained, yielding a total of 50,880 additional cache hits.

Based on these observations, decide how many buffers to add to the cache. In this case, you may make these decisions:

- It is wise to add 250 or 500 buffers, provided memory resources are available. Both of these increments offer significant performance gains.
- It is unwise to add 750 buffers. Nearly the entire performance gain made by such an increase can be made by adding 500 buffers instead. Also, the memory allocated to the additional 250 buffers may be better used by some other Oracle memory structure.
- It is wise to add 1000 buffers, provided memory resources are available. The performance gain from adding 1000 buffers to the cache is significantly greater than the gains from adding 250, 500, or 750 buffers.

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 apply any leftover memory to 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.

The V$CURRENT_BUCKET View

The V$CURRENT_BUCKET view contains the statistics that estimate the performance of a smaller cache. The V$CURRENT_BUCKET view is similar in structure to the V$RECENT_BUCKET view. This table can be accessed only by the user SYS. The following are the columns of the V$CURRENT_BUCKET view:

ROWNUM	The potential number of buffers in the cache.
COUNT	The number of cache hits attributable to buffer number ROWNUM.

The number of rows in this table is equal to the number of buffers in your buffer cache. Each row in the table reflects the number of cache hits attributed to a single buffer. For example, in the second row, the ROWNUM value is 1 and the COUNT value is the number of cache hits for the second buffer. In the third row, the ROWNUM value is 2 and the COUNT value is the number of cache hits for the third buffer.

The first row of the table contains special information. The ROWNUM value is 0 and the COUNT value is the total number of blocks moved into the first buffer in the cache.

Enabling the V$CURRENT_BUCKET View

The collection of statistics in the V$CURRENT_BUCKET view is controlled by the initialization parameter DB_BLOCK_LRU_STATISTICS. The value of this parameter determines whether Oracle collects the statistics. The default value for this parameter is FALSE, which means that the default behavior is not to collect statistics.

To enable the collection of statistics in the V$CURRENT_BUCKET view, set the value of DB_BLOCK_LRU_STATISTICS to TRUE.

Collecting these statistics incurs some performance overhead. To minimize this overhead, collect statistics only when you are tuning the buffer cache; disable the collection of statistics when you are finished tuning.

Querying the V$CURRENT_BUCKET View

From the information in the V$CURRENT_BUCKET view, you can predict the number of additional cache misses that would occur if the number of buffers in the cache were reduced. If your buffer cache currently contains 100 buffers, you may want to know how many more cache misses would occur if it had only 90. To determine the number of additional cache misses, query the V$CURRENT_BUCKET view with the SQL statement:

```
SELECT SUM(count) acm
   FROM V$CURRENT_BUCKET
   WHERE ROWNUM >= 90;
```

You can also determine the hit ratio based on this cache size. Use the following formula to calculate the hit ratio based on the values of the statistics DB BLOCK GETS, CONSISTENT GETS, and PHYSICAL READS and the number of additional cache misses (ACM) returned by the query:

Hit Ratio = 1 - (*physical reads* + ACM / (*db block gets* + *consistent gets*))

Another way to examine the V$CURRENT_BUCKET view is to group the buffers in intervals. For example, if your cache contains 100 buffers, you may want to divide the cache into four 25-buffer intervals. You can query the table with a SQL statement similar to this one:

```
SELECT 25*TRUNC(ROWNUM/25)+1||' to '||25*(TRUNC(ROWNUM/25)+1)
"Interval", SUM(count) "Buffer Cache Hits"
   FROM V$CURRENT_BUCKET
   WHERE ROWNUM > 0 GROUP BY TRUNC(ROWNUM/25);
```

Note that the WHERE clause prevents the query from collecting statistics from the first row of the table. The result of this query might look like

```
Interval          Buffer Cache Hits
--------------- --------------------
1 to 25                       1900
26 to 50                      1100
51 to 75                      1360
76 to 100                      230
```

where:

INTERVAL	Is the interval of buffers in the cache.
BUFFER CACHE HITS	Is the number of cache hits attributable to the buffers in the INTERVAL column.

Examining the query output leads to these observations:

- The last 25 buffers in the cache (buffers 76 to 100) contribute 230 cache hits. If the cache were reduced in size by 25 buffers, 230 cache hits would be lost.
- The third 25-buffer interval (buffers 51 to 75) contributes 1,360 cache hits. If these buffers were removed from the cache, 1,360 cache hits would be lost in addition to the 230 cache hits lost for buffers 76 to 100. Removing 50 buffers would result in losing a total of 1,590 cache hits.
- The second 25-buffer interval (buffers 26 to 50) contributes 1,100 cache hits. Removing 75 buffers from the cache would result in losing a total of 2,690 cache hits.
- The first 25 buffers in the cache (buffers 1 to 25) contribute 1,900 cache hits.

Based on these observations, decide whether to reduce the size of the cache. In this case, you may make these decisions:

- If memory is scarce, it may be wise to remove 25 buffers from the cache. The buffers 76 to 100 contribute relatively few cache hits compared to the total cache hits contributed by the entire cache. Removing 25 buffers will not significantly reduce cache performance, and the leftover memory may be better used by other Oracle memory structures.
- It is unwise to remove more than 25 buffers from the cache. For example, removing 50 buffers would reduce cache performance significantly. The cache hits contributed by these buffers is a significant portion of the total.

Tuning Multiple Buffer Pools

This section covers:

- Overview of the Multiple Buffer Pool Feature
- When to Use Multiple Buffer Pools
- Tuning the Buffer Cache Using Multiple Buffer Pools
- Enabling Multiple Buffer Pools
- Using Multiple Buffer Pools
- Dictionary Views Showing Default Buffer Pools
- How to Size Each Buffer Pool
- How to Recognize and Eliminate LRU Latch Contention

Overview of the Multiple Buffer Pool Feature

Schema objects are referenced with varying usage patterns; therefore, their cache behavior may be quite different. Multiple buffer pools enable you to address these differences. You can use a “keep” buffer pool to maintain an object in the buffer cache, and a “recycle” buffer pool to prevent an object from taking up 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 cache for objects that have not been assigned to one of the buffer pools.

Each buffer pool in Oracle8 comprises a number of working sets. A different number of sets can be allocated for each buffer pool. All sets use the same LRU replacement policy. A strict LRU aging policy provides very good hit rates in most cases, but you can sometimes improve the hit rate 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 cache and should reuse buffers more quickly than the default cache.

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 or not 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>';
   ```

   Since two objects can have the same name (if they are different types of object), 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 x$bh WHERE obj = <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*.

   $$\%\ \text{cache used by } segment\_name = \frac{buffers \quad \text{(Step 2)}}{total\ buffers \quad \text{(Step 3)}}$$

   Note: This technique works only for a single segment; for a partitioned object, the query must be run for each partition.

If the number of local block gets equals the number of physical reads for statements involving such objects, consider employing 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 keep that object in the buffer cache so that the next use of the block will not require another disk I/O operation. Conversely, if a block probably will not be reused within a reasonable period of time, there is no reason to keep it in the cache; the block should be discarded to make room for a more popular block.

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. Between instances a buffer pool may be different sizes or not defined at all. Each instance should be tuned 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 of the buffer pools cannot, therefore, exceed this value. Likewise, the number of LRU latches allocated to each buffer pool is taken from the total number allocated to the instance by the DB_BLOCK_LRU_LATCHES parameter. If either constraint is violated then an error occurs and the database is not mounted.

The minimum number of buffers that 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.

Oracle8 defines three buffer pools: KEEP, RECYCLE, and DEFAULT. The default buffer pool always exists. It is equivalent to the single buffer cache in Oracle7. You do not explicitly define the size of the default buffer pool and 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 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 will go in the specified buffer pool.

The BUFFER_POOL clause is used to define the default buffer pool for an object. This clause is valid for CREATE and ALTER table, cluster, and index DDL statements. The buffer pool name is case insensitive. The blocks from an object without an explicitly set buffer pool go into the DEFAULT buffer pool.

If a buffer pool is defined for a partitioned table or index then each partition of the object inherits the buffer pool from the table or index definition unless overridden with a specific buffer pool.

When the default buffer pool of an object is changed using the ALTER statement, all buffers that currently contain 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 will go into the new buffer pool.

The syntax 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 a simple alter table as well as modify partition, move partition, add partition, and split partition (for both new partitions).

- ALTER INDEX *index_name* ... STORAGE (*buffer_pool_clause)*

  A buffer pool can be defined during a simple alter index as well as rebuild, modify partition, split partition (for both new partitions), and rebuild partition.

- ALTER CLUSTER *cluster_name* ... STORAGE (*buffer_pool_clause)*

Dictionary Views Showing Default Buffer Pools

The following dictionary views have a BUFFER POOL column, which indicates 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.

How to Size 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:

$$hit\ ratio = 1 - \frac{physical\ reads}{(block\ gets + consistent\ gets)}$$

where the values of physical reads, block gets, and consistent gets can be obtained for the keep buffer pool from the following query:

```
SELECT PHYSICAL_READS, BLOCK_GETS, CONSISTENT_GETS
FROM V$BUFFER_POOL_STATISTICS WHERE NAME = 'KEEP';
```

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 has been running for a while and has achieved steady-state performance. Calculate the hit ratio by taking two snapshots of system performance using the above query and using the delta values of physical reads, block gets, and consistent gets.

Keep in mind that a 100% buffer pool hit ratio may not be necessary. Often you can decrease the size of your keep buffer pool by quite a bit and still maintain a sufficiently high hit ratio. Those blocks can be allocated 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.

Remember, 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 being available for other, more active blocks.

Recycle Buffer Pool

The goal of the recycle buffer pool is to eliminate blocks from memory as soon as they are no longer needed. If an application accesses the blocks of a very large object in a random fashion then there is little chance of reusing a block stored in the buffer pool before it is aged out. This is true regardless of the size of the buffer pool (given the constraint of the amount of available physical memory). Because of this, the object's blocks should not be cached; those cache buffers can be allocated to other objects.

Be careful, however, not to discard blocks from memory too quickly. If the buffer pool is too small then it is possible for a block to 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 tuple. If the block is removed from the cache after the select statement then it must be read from disk again to perform the update. The block needs to 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 is

given 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 high then the pool is probably too small. Likewise, the number of “log file sync” wait events will increase. One way to size the recycle buffer pool is to run the system with the recycle buffer pool disabled. At steady state the number of buffers in the default buffer pool that are being consumed by segments that would normally go in the recycle buffer pool can be divided by four. That number can be used to size 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 look at 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 heated and the segment may be a good candidate for the recycle cache. If the ratio is high (perhaps 3) then buffers are frequently heated and the segment might be a good candidate for the keep cache.

How to Recognize and Eliminate 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 also 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

and

number_of_buffers / 50

This is because no set can have fewer than 50 buffers. If you specify a value larger than the maximum, then the number of latches is automatically reset 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 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.

Note that 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 19, "Tuning Parallel Execution"

Reallocating Memory

After resizing your Oracle memory structures, reevaluate the performance of the library cache, the data dictionary cache, and the buffer cache. If you have reduced the memory consumption of any one of these structures, you may want to allocate more memory to another structure. For example, if you have reduced the size of your buffer cache, you may now want to take advantage of the additional available memory by using it 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 in order 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 (request) message 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 most 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 multithreaded server (MTS) represents a compromise solution that is highly effective at reducing the number of operating system processes on the server, but less effective in reducing the overall memory requirement. Use of the MTS has no effect on the number of network connections.

Shared connections are possible in an Oracle Forms environment by using an intermediate server that is also a client, and using 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. This in turn owns a connection to the central server.

15

Tuning I/O

This chapter explains how to avoid I/O bottlenecks that could prevent Oracle from performing at its maximum potential. It covers the following topics:

- Understanding I/O Problems
- How to Detect I/O Problems
- How to Solve 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

Understanding I/O Problems

This section introduces I/O performance issues. It covers:

- Tuning I/O: Top Down and Bottom Up
- Analyzing I/O Requirements
- Planning File Storage
- Choosing Data Block Size
- Evaluating Device Bandwidth

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 14, "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 perform the necessary I/O as efficiently as possible.

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 distinguish the objects against which each operation is performed.

   In an OLTP application, for example, each transaction might involve:

   - 1 read from object A
   - 1 read from object B
   - 1 write to object C

   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 transactions per second (or "tps") which 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 performance.

   To do this, ascertain the number of I/Os that each disk can perform per second. This will depend 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 15–1 Relative Disk Speed

Disk Speed:	File System	Raw Devices
Reads per second	fast	slow
Writes per second	slow	fast

Lay out in a table like this the relative speed per operation of your disks:

Table 15–2 Disk I/O Analysis Worksheet

Disk Speed:	File System	Raw Devices
Reads per second		
Writes per second		

The disks in the current example have the following characteristics:

Table 15–3 Sample Disk I/O Analysis

Disk Speed:	File System	Raw Devices
Reads per second	50	45
Writes per second	20	50

4. Figure the number of disks needed to achieve your I/O performance target. Use a table like this:

Table 15–4 Disk I/O Requirements Worksheet

Object	If Stored on File System			If Stored on Raw Devices		
	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 15–5 shows the values from this example.

Table 15–5 Sample Disk I/O Requirements

Object	If Stored on File System			If Stored on Raw Devices		
	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 on the disks you have available, if you store the data on raw devices, block devices, or 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.

Identifying the Required Read/Write Operations

Evaluate your application to determine how often it requires each type of I/O operation. Table 15–6 shows the types of read and write operations performed by each of the background processes, by foreground processes, and by parallel query slaves.

Table 15–6 Read/Write Operations Performed by Oracle Processes

	Process							
Operation	**LGWR**	**DBW*n***	**ARCH**	**SMON**	**PMON**	**CKPT**	**Fore-ground**	**PQ Slave**
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. (Note also that many concurrent processes may generate overhead due to head and arm movement of the disk drives.)

Attention: The figures provided in this example *do not* constitute a rule of thumb. They were generated by an actual UNIX-based test system using particular disks. *These figures will differ significantly for different platforms and different disks!* To make accurate judgments, you must *test your own system* using an approach like the one demonstrated in this section. Alternatively, contact your system vendor for information on relative disk performance for the different operations.

Table 15–7 and Figure 15–1 show speed of sequential read in milliseconds per I/O, for each of the three disk layout options on a test system.

Table 15–7 Block Size and Speed of Sequential Read (Sample Data)

Block Size	Speed of Sequential Read on:		
	Raw Device	**Block Device**	**UNIX File System**
512 bytes	1.4	0.6	0.4
1K	1.4	0.6	0.3
2K	1.5	1.1	0.6
4K	1.6	1.8	1.0
8K	2.7	3.0	1.5
16K	5.1	5.3	3.7
32K	10.1	10.3	8.1
64K	20.0	20.3	18.0
128K	40.4	41.3	36.1
256K	80.7	80.3	61.3

Doing research like this helps you pick the right stripe size. In this example, it takes at most 5.3 milliseconds to read 16 K. If your data were in chunks of 256 K, you could stripe the data over 16 disks (as described on page 15 - 23) and maintain this low read time. By contrast, if all the 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 15–1 Block Size and Speed of Sequential Read (Sample Data)

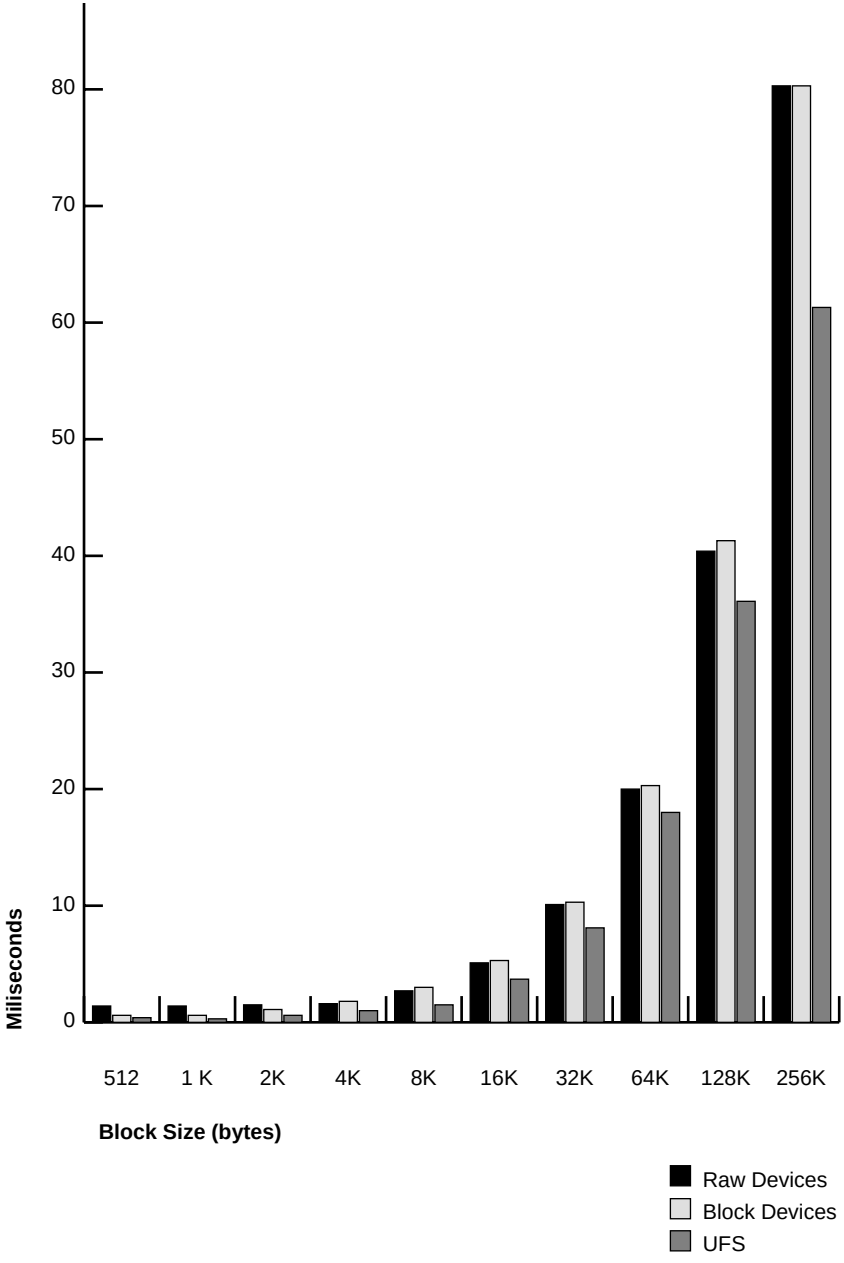


Table 15–8 and Figure 15–2 show speed of sequential write in milliseconds per I/O, for each of the three disk layout options on the test system.

Table 15–8 Block Size and Speed of Sequential Write (Sample Data)

Block Size	Speed of Sequential Write on		
	Raw Device	**Block Device**	**UNIX File System**
512 bytes	11.2	11.8	17.9
1K	11.7	11.9	18.3
2K	11.6	13.0	19.0
4K	12.3	13.8	19.8
8K	13.5	13.8	21.8
16K	16.0	27.8	35.3
32K	19.3	55.6	62.2
64K	31.5	111.1	115.1
128K	62.5	224.5	221.8
256K	115.6	446.1	429.0

Figure 15–2 Block Size and Speed of Sequential Write (Sample Data)

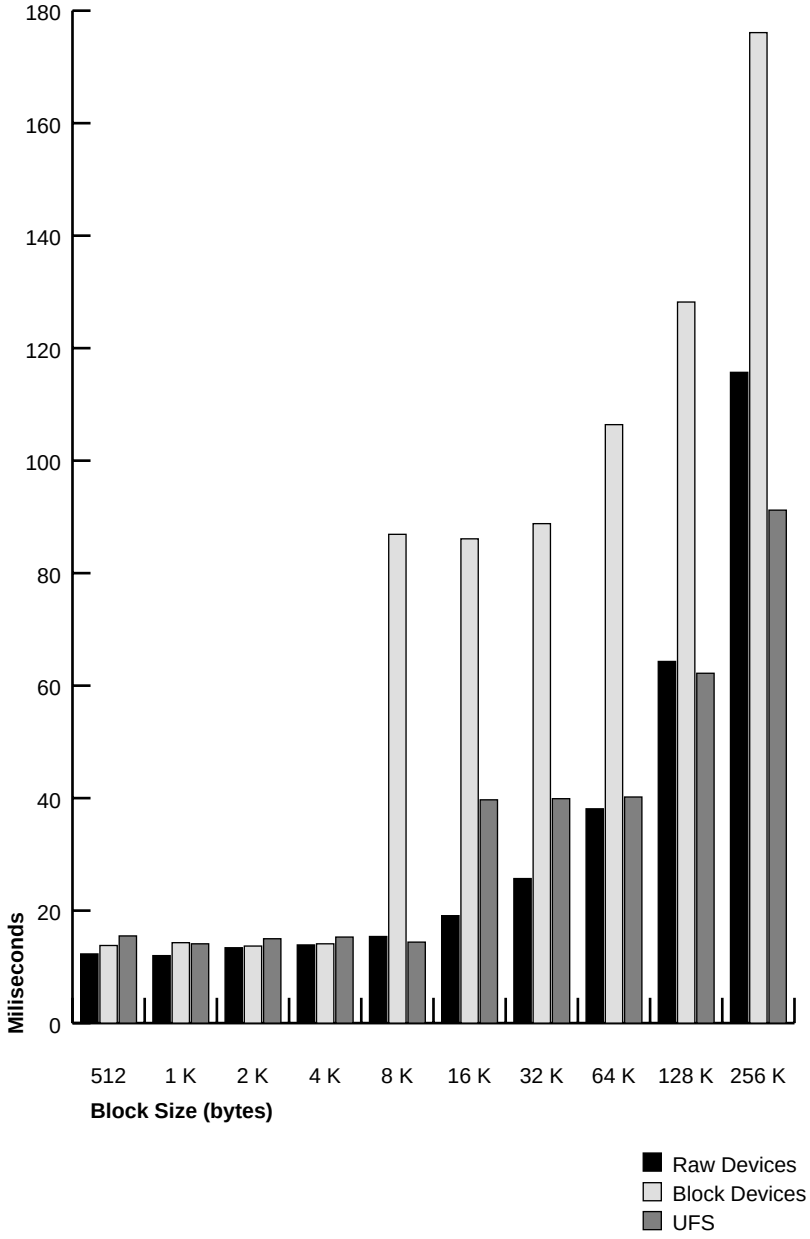


Table 15–9 and Figure 15–3 show speed of random read in milliseconds per I/O, for each of the three disk layout options on the test system.

Table 15–9 Block Size and Speed of Random Read (Sample Data)

Block Size	Speed of Random Read on		
	Raw Device	**Block Device**	**UNIX File System**
512 bytes	12.3	13.8	15.5
1K	12.0	14.3	14.1
2K	13.4	13.7	15.0
4K	13.9	14.1	15.3
8K	15.4	86.9	14.4
16K	19.1	86.1	39.7
32K	25.7	88.8	39.9
64K	38.1	106.4	40.2
128K	64.3	128.2	62.2
256K	115.7	176.1	91.2

Figure 15–3 Block Size and Speed of Random Read (Sample Data)

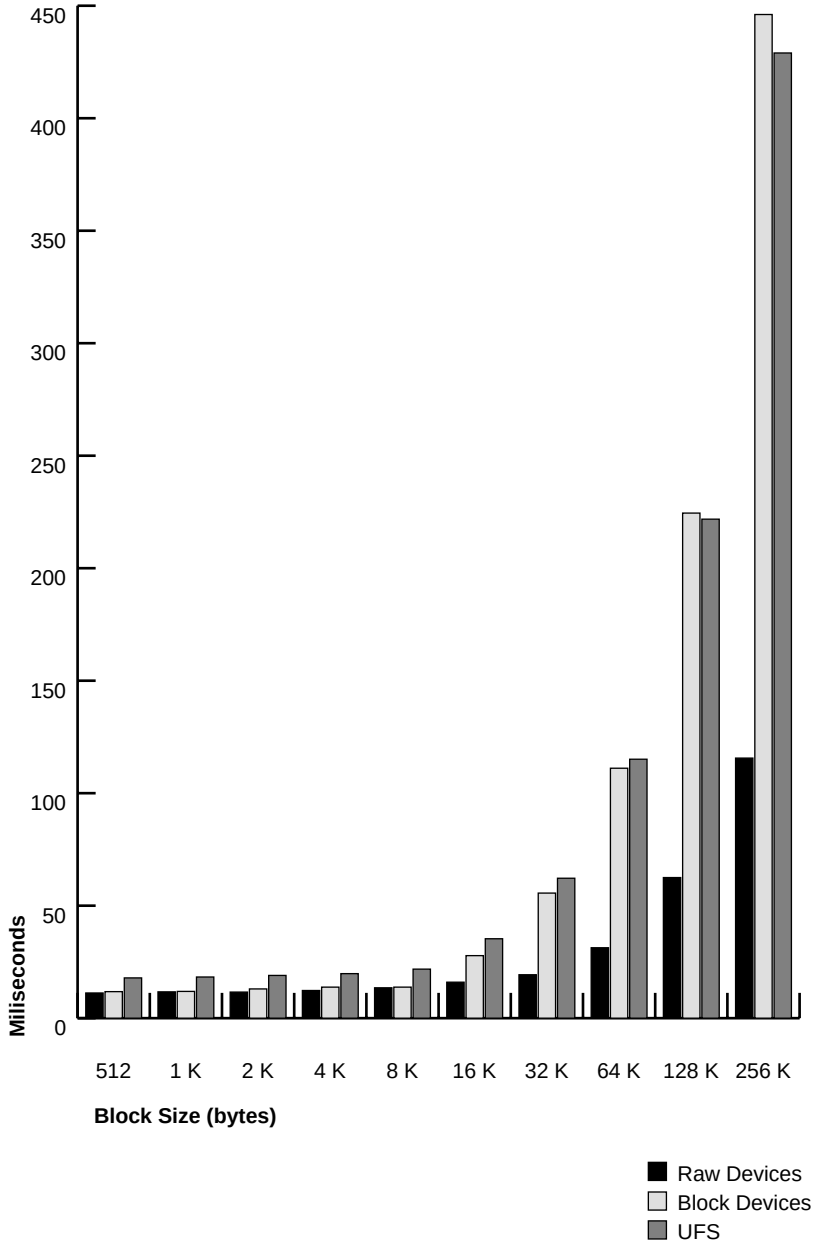
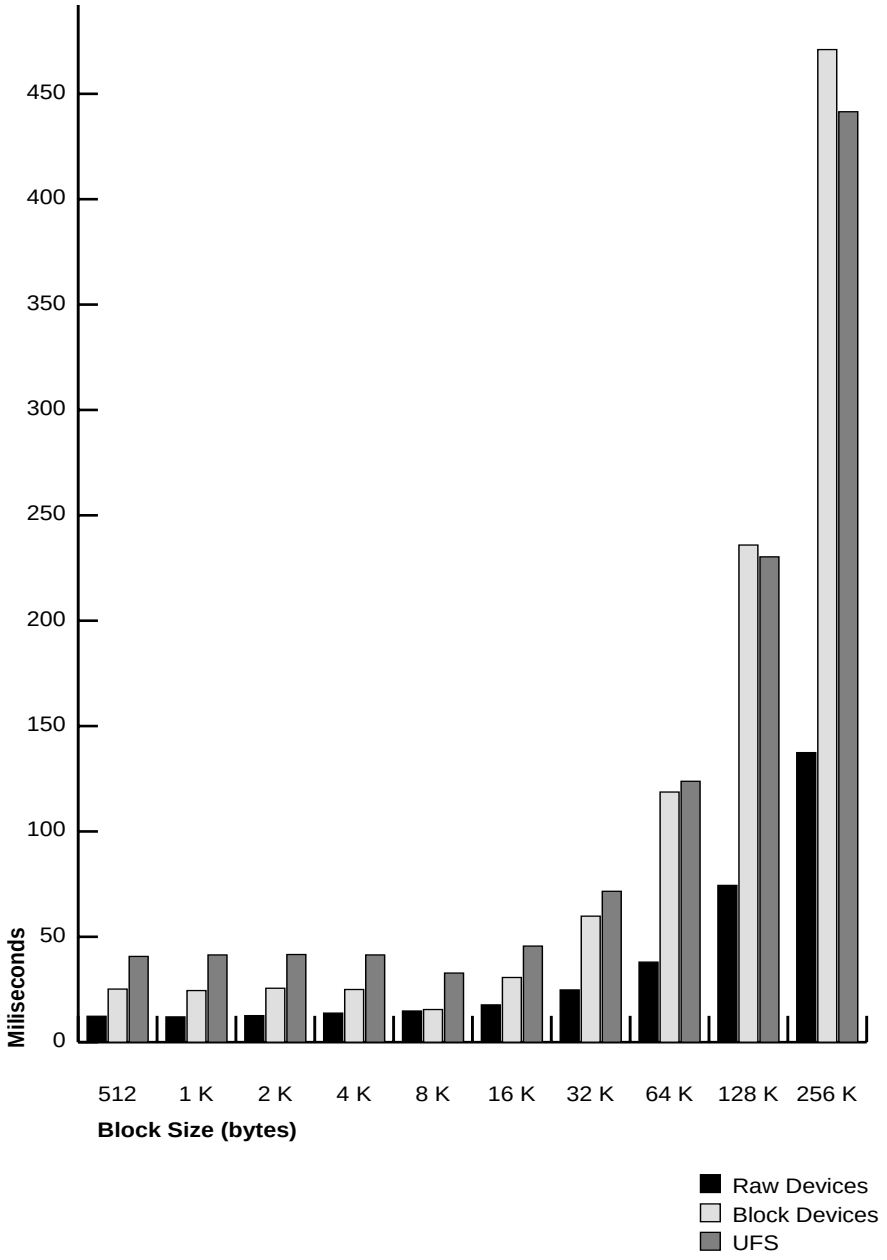


Table 15–10 and Figure 15–4 show speed of random write in milliseconds per I/O, for each of the three disk layout options on the test system.

Table 15–10 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
1K	12.0	24.5	41.4
2K	12.6	25.6	41.6
4K	13.8	25.0	41.4
8K	14.8	15.5	32.8
16K	17.7	30.7	45.6
32K	24.8	59.8	71.6
64K	38.0	118.7	123.8
128K	74.4	235.9	230.3
256K	137.4	471.0	441.5

Figure 15–4 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), 8K would be a good block size. Raw devices and UNIX file system 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 8K block size.

Attention: *Figures shown in the preceding example will differ significantly on different platforms, and with different disks!* To plan effectively you must test I/O performance on your own system!

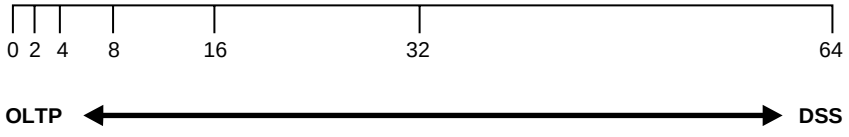
Choosing Data Block Size

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), for best performance you want to get all the desired data from a single block in one read. How you store the data determines whether or not 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" on page 15-6). This assumes that the operating system can buffer the I/O so that the next block will be read from that particular buffer.

Figure 15–5 illustrates the suitability of various block sizes to online transaction processing (OLTP) or decision support (DSS) applications.

Figure 15–5 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 15–11 Block Size Advantages and Disadvantages

Block Size	Advantages	Disadvantages
Small (2K-4K)	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 (8K)	If rows are of medium size, you can bring a number of rows into the buffer cache with a single I/O. With 2K or 4K 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 8K block size and 50B row size, you would be wasting 7,950B in the buffer cache when doing random access.
Large (16K-32K)	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.

How to Detect I/O Problems

If you suspect a problem with I/O usage, you must evaluate two areas:

- Checking System I/O Utilization
- Checking Oracle I/O Utilization

Oracle compiles file I/O statistics that reflect disk access to database files. Note, however, that these statistics report only the I/O utilization of Oracle sessions—yet every process running on your system affects the available I/O resources. Tuning non-Oracle factors can thus result in better Oracle 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 table.

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.

Attention: For information on other platforms, please check your operating system documentation.

See Also: Oracle platform-specific 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 Tables Contain I/O Statistics

Table 15–12 shows dynamic performance tables to check for I/O statistics relating to Oracle database files, log files, archive files, and control files.

Table 15–12 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 15–13 lists processes whose statistics reflect I/O throughput for the different Oracle file types.

Table 15–13 File Throughput Statistics for Oracle Processes

	Process							
File	LGWR	DBW*n*	ARCH	SMON	PMON	CKPT	Fore-ground	PQ Slave
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. See if you can 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 15-5.

How to Check Oracle Datafile I/O with V$FILESTAT

Examine disk access to database files through the dynamic performance table 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 table 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 table 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.

How to Solve 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 table 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 18-13.

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. Since 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 15-23.

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:

- What Is Striping?
- I/O Balancing and Striping
- How to Stripe Disks Manually
- How to Stripe Disks with Operating System Software
- How to Do Hardware Striping with RAID

What Is Striping?

"Striping" is the practice of dividing a large table's data into small portions and storing 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). Since redo logs are written serially, any drive dedicated to redo log activity will normally require very little head movement. This will significantly speed up 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.

Note that 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 member is finished. To avoid I/O problems, striping should be done on the same number of disks as the data itself.

For example, if you have 160K 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 160K 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.

How to Stripe Disks Manually

To stripe disks manually, you need to relate the object's storage requirements to its I/O requirements.

1. Begin by evaluating the object's disk storage requirements. You need to know
   - the size of the object
   - the size of the disk

   For example, if the object requires 5G in Oracle storage space, you would need one 5G disk or two 4G disks to accommodate it. On the other hand, if the system is configured with 1G or 2G 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 15-3. You must take the larger of the storage requirement and the I/O requirement.

   For example, if the storage requirement is 5 disks (1G 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 command. 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 command. 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 1G (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: *Oracle8 SQL Reference* for more information on MINEXTENTS and the other storage parameters.

How to Stripe Disks with Operating System Software

An alternative to striping disks manually, you can use operating system striping software, such as a logical volume manager (LVM), to stripe disks. With striping software, the biggest concern is choosing the right stripe size. This depends on Oracle block size and type of disk access.

Table 15–14 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 big you can end up with a hot spot on one disk or on a small number of disks. You can avoid this problem by making the stripe size smaller, 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 access is needed only to rows 35 through 55, then only 2 disks must handle all the I/O. At this rate, the system cannot achieve the desired level of performance.

You can correct this problem by spreading the rows of interest 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 of interest over all the disks and I/O throughput would be much improved.

How to Do Hardware Striping 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 an object such as a table or rollback segment is created, space is allocated in the database for the data. This space is called a *segment*. If subsequent database operations cause the data to grow and exceed the space allocated, Oracle extends the segment. Dynamic extension can reduce 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 table V$SYSSTAT. By default, this table is available only to the user SYS and to users granted the SELECT ANY TABLE system privilege, such as SYSTEM. Monitor this statistic over some period of time while your application is running, with this query:

```
SELECT name, value
    FROM v$sysstat
    WHERE name = 'recursive calls';
```

The output of this query might look like this:

```
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 one of the activities that generate recursive calls other than dynamic extension. If you determine that these recursive calls are caused by dynamic extension, you should try to 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 how much space to allow for a table, see the *Oracle8 Administrator's Guide*.
2. Choose storage parameter values so that Oracle allocates extents large enough to accommodate all of 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 few large extents or many small extents, consider the benefits and drawbacks of each in light of your plans for the growth and use of your tables.

Automatically resizable datafiles can also cause a problem 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 the 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 operating system. Accordingly, this affects the number of extents which can be stored in a single operating system block. Typically, this value is as follows:

Table 15–15 Block Size and Maximum Number of Extents (Typical Values)

Block Size (K)	Max. Number of Extents
2	121
4	255
8	504
16	1032
32	2070

For best performance, you should be able to read the extent map with a single I/O. Performance will degrade if multiple I/Os are necessary for a full table scan to get the extent map.

Furthermore, a large number of extents can degrade data dictionary performance. Performance would suffer, for example, if you had 8,000 extents, and had to bring them all into the dictionary cache.

Evaluating Multiple Extents

This section explains various ramifications of the use of multiple extents.

You cannot put very 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. Note also:

- 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 materially affect the performance of full scans on an operational 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 command 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
```

Warning: 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 OPTIMAL accordingly, see *Oracle8 Administrator's Guide*.

For Long Queries

Assign large rollback segments to transactions that modify data which 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. Such transactions generate 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 to 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.

You can identify migrated and chained rows in a table or cluster by using the ANALYZE command with the LIST CHAINED ROWS option. This command 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 may vary depending on your operating system. Your output table must have the same column names, datatypes, and sizes as the CHAINED_ROWS table.

To reduce migrated and chained rows in an existing table, follow these steps:

1. Use the ANALYZE command 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 command 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.

You can detect migrated or chained rows by checking the "table fetch continued row" statistic 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 (re-create) tables and indexes with a high deletion rate.

Note: PCTUSED is not the opposite of PCTFREE; it concerns space management.

See Also: *Oracle8 Concepts* for more information.

Modifying the SQL.BSQ File

The SQL.BSQ file is run when you issue the CREATE DATABASE statement. This file contains the actual table definitions that make up the Oracle Server. The views that you use as a DBA are based on these tables. Oracle Corporation recommends that users strictly limit their 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, you should not decrease the setting of a storage parameter to a value below the default. (If the value of MAXEXTENTS is large, you can make PCTINCREASE small—or even 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, any modifications you make will not be carried forward when the dictionary is migrated to later releases.

See Also: *Oracle8 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 to disk affect performance. If the sort area size is too big, too much memory may be utilized; if the sort area size is too small, many sorts to disk may result, with correspondingly worse performance. This section describes:

- Sorting to Memory
- If You Do Sort to Disk
- Optimizing Sort Performance with Temporary Tablespaces
- Using NOSORT to Create Indexes Without Sorting
- GROUP BY NOSORT
- Optimizing Large Sorts with SORT_DIRECT_WRITES

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: *Oracle8 Concepts* lists SQL statements that perform sorts.

Recognizing Large Sorts

Oracle collects statistics that reflect sort activity and stores them in the dynamic performance table V$SYSSTAT. By default, this table 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 a period of time while your application is running:

```
SELECT name, value
   FROM v$sysstat
   WHERE name IN ('sorts (memory)', 'sorts (disk)');
```

The output of this query might look like this:

```
NAME                                                     VALUE
-------------------------------------------------------- ----------
sorts(memory)                                               965
sorts(disk)                                                   8
```

The information in V$SYSSTAT can also be obtained through the Simple Network Management Protocol (SNMP).

Increasing SORT_AREA_SIZE to Avoid Sorting to Disk

SORT_AREA_SIZE is a dynamically modifiable initialization parameter that specifies the maximum amount of program global area (PGA) memory to use for a sort. There is a trade-off between SORT_AREA_SIZE and PGA.

If a significant number of sorts require disk I/O to temporary segments, then your application's performance may benefit from increasing the size of the sort area. In this case, increase the value of SORT_AREA_SIZE. The maximum value of this parameter depends on your operating system. You need to determine how large a SORT_AREA_SIZE makes sense. If it is big enough, most sorts should not go to disk (unless, for example, you are sorting a 10 gigabyte table). If the sort does *not* go to disk, you have the option of writing or not writing to the buffer cache.

See Also: "Optimizing Large Sorts with SORT_DIRECT_WRITES" on page 15-40
"SORT_AREA_SIZE" on page 19-9

Performance Benefits of Large Sort Areas

Increasing the sort area increases the size of each run and decreases the total number of runs. Reducing the total number of runs may reduce the number of merges Oracle must perform to obtain the final sorted result.

Performance Trade-offs for Large Sort Areas

Increasing the size of the sort area causes each Oracle process that sorts to allocate more memory. This increase reduces the amount of memory available 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 retained size of the sort area, or the size to which Oracle reduces the sort area if its data is not expected to be referenced soon. To do this, decrease the value of the SORT_AREA_RETAINED_SIZE parameter. 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.

If You Do Sort to Disk

If you do 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 *Oracle8 Concepts* for more information on PCTINCREASE.)

Optimizing Sort Performance with Temporary Tablespaces

You can optimize sort performance by specifying a tablespace as TEMPORARY upon creation (or subsequently altering that tablespace) and performing the sort in that tablespace. Normally, a sort may require many space allocation calls to allocate and deallocate temporary segments. If a tablespace is specified as TEMPORARY, one sort segment in that tablespace is cached and used for each instance requesting a sort operation. This scheme bypasses the normal space allocation mechanism and can greatly improve performance of medium-sized sorts that cannot be done completely in memory.

To specify a tablespace as temporary, use the TEMPORARY keyword of the CREATE TABLESPACE or ALTER TABLESPACE commands. TEMPORARY cannot be used with tablespaces that contain permanent objects (such as tables or rollback segments).

The temporary tablespace should be striped over many disks, preferably with some operating system striping tool. For example, if the temporary tablespace is only striped over 2 disks with a maximum of 50 I/Os per second each, then you can only do 100 I/Os per second. This restriction could become a problem, making sort operations take a very long time. You could speed up sorts fivefold if you were to stripe the temporary tablespace over 10 disks. This would enable 500 I/Os per second.

Change the SORT_READ_FAC parameter, which is a ratio describing the amount of time necessary to read a single database block divided by the block transfer rate. The value is operating system specific; the default value is typically 5, but the parameter should usually be set to 16 or 32. This allows the system to read more blocks per pass from a temporary table. For temporary tablespaces, SORT_READ_FAC plays a role similar to the parameter DB_FILE_MULTIBLOCK_READ_COUNT.

See Also: *Oracle8 SQL Reference* for more information about the syntax of the CREATE TABLESPACE and ALTER TABLESPACE commands.

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 the 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 command. 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.

Optimizing Large Sorts with SORT_DIRECT_WRITES

If memory and temporary space are abundant on your system, and you perform many large sorts to disk, you can set the initialization parameter SORT_DIRECT_WRITES to increase sort performance.

Behavior of SORT_DIRECT_WRITES

When this parameter is set to TRUE, each sort allocates several large buffers in memory for direct disk I/O. You can set the initialization parameters SORT_WRITE_BUFFERS and SORT_WRITE_BUFFER_SIZE to control the number and size of these buffers. The sort will write an entire buffer for each I/O operation. The Oracle process performing the sort writes the sort data directly to the disk, bypassing the buffer cache.

The default value of SORT_DIRECT_WRITES is AUTO. When the parameter is unspecified or set to AUTO, Oracle automatically allocates direct write buffers if the SORT_AREA_SIZE is at least ten times the minimum direct write buffer configuration.

The memory for the direct write buffers is subtracted from the sort area, so the total amount of memory used for each sort is still SORT_AREA_SIZE. Setting SORT_WRITE_BUFFERS and SORT_WRITE_BUFFER_SIZE has no effect when SORT_DIRECT_WRITES is AUTO.

Performance Trade-offs of Direct Disk I/O for Sorts

Setting SORT_DIRECT_WRITES to TRUE causes each Oracle process that sorts to allocate memory in addition to that already allocated for the sort area. The additional memory allocated is calculated as follows:

SORT_WRITE_BUFFERS * SORT_WRITE_BUFFER_SIZE

The minimum direct write configuration on most platforms is two 32K buffers (2 * 32K), so direct write is generally allocated only if the sort area is 640K or greater. With a sort area smaller than this, direct write will not be performed.

Ensure that your operating system has enough free memory available to accommodate this increase. Also, sorts that use direct writes tend to consume more temporary segment space on disk.

One way to avoid increasing memory usage is to decrease the sort area by the amount of memory allocated for direct writes. Note that reducing the sort area may increase the number of sorts to disk, which will decrease overall performance. A good rule of thumb is that the total memory allocated for direct write buffers should be less than one-tenth of the memory allocated for the sort area. If the mini-

mum configuration of the direct write buffers is greater than one-tenth of your sort area, then you should not trade sort area for direct write buffers.

Tuning Checkpoints

A checkpoint is an operation that Oracle performs automatically. Checkpoints can momentarily reduce performance. This section explains:

- How Checkpoints Affect Performance
- Choosing Checkpoint Frequency
- Reducing the Performance Impact of a Checkpoint

How Checkpoints Affect Performance

Checkpoints affect:

- recovery time performance
- run-time performance

Frequent checkpoints can reduce 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 recovery.

However, a checkpoint can momentarily reduce run-time performance for these reasons:

- Checkpoints cause DBW*n* processes to perform I/O.
- If CKPT is not enabled, checkpoints cause LGWR to update datafiles and may momentarily prevent LGWR from writing redo entries.

The overhead associated with a checkpoint 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 fast recovery time than run-time performance, choose a higher checkpoint frequency.

Because checkpoints on log switches 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 0. This value eliminates time-based checkpoints.

Such settings eliminate all checkpoints except those that occur on log switches.

You can further reduce checkpoints by reducing the frequency of log switches. To reduce log switches, increase the size of your redo log files so that the files do not fill as quickly.

Reducing the Performance Impact of a Checkpoint

To reduce the performance impact of checkpoints, make sure that DBW*n* process(es) write enough during periods of normal (nonpeak) activity. DBW*n* activity sometimes has a pattern of sharp peaks and valleys. If DBW*n* is tuned to be more aggressive in writing, then the average level of its activity will be raised, and it will not fall behind.

DB_BLOCK_CHECKPOINT_BATCH specifies the number of blocks the DBW*n* process(es) can write out in one batch as part of a checkpoint. If the internal Oracle write batch size is 512 buffers, and DB_BLOCK_CHECKPOINT_BATCH is set to 8, then the checkpoint may take a very long time. Because you only write 8 blocks at a time, the start and end of a checkpoint will take a long time, going through 512 buffers, 8 at a time.

Note that only current or consistent read blocks are checkpoints. By contrast, sort blocks are *not* checkpointed.

See Also: *Oracle8 Concepts* for a complete discussion of checkpoints.

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 a lot of 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.

Tuning DBWn I/O

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 which have a large number of CPUs. Note that 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 write batch too large may result in very 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.
5. Adjust the current write batch size so that it equals the number of writes you need plus the number of writes needed for checkpointing (specified by DB_BLOCK_CHECKPOINT_BATCH). You can calculate the desired internal write batch size (IWBS) as follows:

   newIWBS = DB_BLOCK_CHECKPOINT_BATCH + *currentIWBS*

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 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.

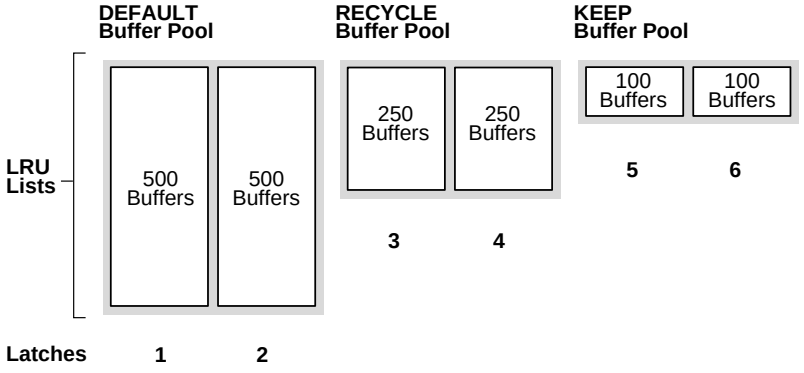
For example, if you have 2 database writer processes and 4 LRU lists (and thus 4 latches), the DBW*n* processes obtain latches in a round-robin fashion. DBW0 obtains latch 1, DBW1 obtains latch 2, then DBW2 obtains latch 3 and DBW3 obtains latch 4. Similarly, if your system has 8 CPUs and 3 DBW*n* processes, you would want to 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, too, that when there are multiple buffer pools, each buffer pool has a contiguous range of LRU latches.)

Consider an example in which 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 15–6 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

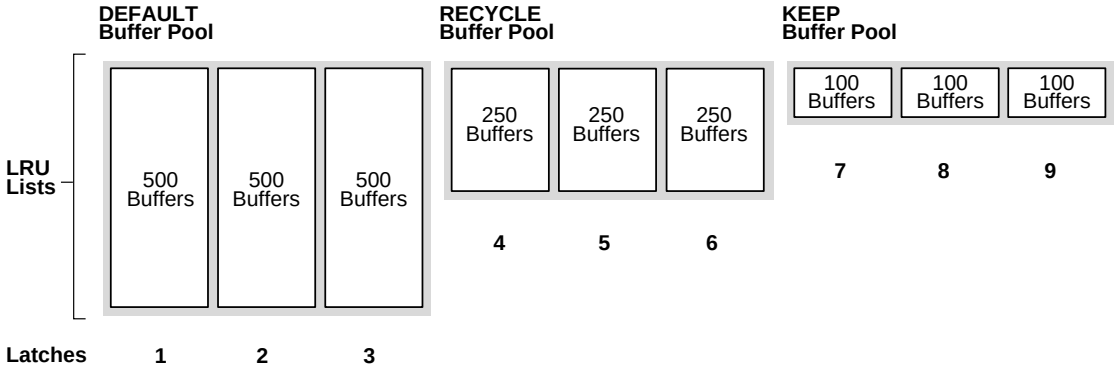
Thus the load carried by each of the DBW*n* processes differs, and performance suffers. If, however, there are 3 latches in each pool, the DBW*n* processes have equal loads and performance is optimized.

Note in particular that the different buffer pools have different rates of block replacement. Ordinarily, blocks are rarely modified in the KEEP pool and fre-

quently modified in the RECYCLE pool; this means that you need to write out blocks more frequently from the RECYCLE pool than from the KEEP pool. As a result, owning 100 buffers from one pool is not the same as owning 100 buffers from the other pool. To be perfectly load balanced, each DBW*n* process should have the same number of LRU lists from each type of buffer pool.

A well configured system might have 3 DBW*n* processes and 9 latches, with 3 latches in each buffer pool

Figure 15–7 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 750
DBW1 gets latch 2 (500) and latch 5 (250) and latch 8 (100) for 750
DBW2 gets latch 3 (500) and latch 6 (250) and latch 9 (100) for 750

Configuring the Large Pool

You can optionally configure the large pool so that Oracle has a separate pool from which it can request large memory allocations. This prevents competition with other subsystems for the same memory. As Oracle allocates more shared pool memory for the multithreaded server (MTS) 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. Note that the large pool does not have an LRU list; Oracle will not attempt to age memory out of the large pool.

Use the following parameters to configure the large pool:

- LARGE_POOL_MIN_ALLOC
- LARGE_POOL_SIZE

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: *Oracle8 Concepts* for further information about the large pool.

Oracle8 Reference for complete information about initialization parameters.

16

Tuning Networks

This chapter introduces networking issues that affect tuning. Topics in this chapter include

- How to Detect Network Problems
- How to Solve Network Problems

How to Detect 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.

Measuring the amount of delay the network adds to performance can be difficult. Three dynamic performance views are useful in this regard: 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.

How to Solve 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 *Oracle 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 multithreaded 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 see if 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: *Oracle 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 multithreaded 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.

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.

Connection Manager you control client access to dedicated servers. In addition, it provides multiple protocol support so that a client and server with different networking protocols can communicate with each other.

See Also: *Oracle Net8 Administrator's Guide*

17

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
- How to Detect Operating System Problems
- How to Solve Operating System Problems

Understanding Operating System Performance Issues

- Overview
- Operating System and Hardware Caches
- Raw Devices
- Process Schedulers

Overview

Operating system performance issues commonly involve process management, memory management, and scheduling. If you have tuned the Oracle instance performance and still need faster performance, you should 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 big effect on application performance. Sometimes there is simply not a lot of room for improvement on the operating system side. Changes in the Oracle configuration or in the application itself are likely to make a bigger difference in operating system efficiency than changing the O/S directly.

For example, if your application gives rise to a lot of 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 go down. 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 which do not directly conflict with Oracle's own cache management, but which can consume resources with no benefit to the user. This occurrence is most marked on a UNIX system with the database container files held in the UNIX file store: by default all database I/O will go through the file system cache. On some UNIX-based 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 resource and allows the file system cache to be dedicated to nondatabase 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. They involve a lot of work on the part of the DBA, but may provide some performance benefit.

Raw devices impose a penalty on full table scans, but may be essential on UNIX-based systems if the UNIX implementation does not support write through cache. The UNIX file system speeds up full table scans by reading ahead when the server starts requesting contiguous data blocks. It also caches full table scans. If your UNIX-based system does not support the write-through option on writes to the file system, then it is essential to use raw devices to ensure that at commit and checkpoint, the data which the server assumes is safely established on disk has actually gotten there. If this is not the case, then recovery from a UNIX or 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 (“threads” on NT systems) are involved in the operation of Oracle, and they all access the shared memory resources in the SGA.

Be sure that all Oracle processes, both background processes 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, and release the resource.

How to Detect 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)

Look at CPU utilization to see the ratio between time spent running in application mode and 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.

How to Solve Operating System Problems

This section provides hints for tuning various systems.

- Performance on UNIX-Based Systems
- Performance on NT Systems
- Performance on Mainframe Computers

Familiarize yourself with platform-specific issues, so that you know what performance options your operating system provides. For example, some platforms have post wait drivers, which 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-based systems, try to find a good ratio between the amount of time the operating system runs (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 you should investigate to determine what is wrong.

The ratio of time spent in each mode is only a symptom of the underlying problem, which might have to do with:

- swapping
- executing too many O/S system calls
- running too many processes

If such conditions exist, then there is less time available for the application to run. The more time you can release from the operating system side, the more transactions your application can perform.

Performance on NT Systems

On NT systems, as with UNIX-based systems, you should establish an appropriate ratio between time in application mode and time in system mode. On NT you can easily monitor many factors with Performance Monitor: CPU, network, I/O, and memory are all displayed on the same graph, to assist you in avoiding bottlenecks in any of these areas. Note that the term “process” as used in this tuning manual refers to a “thread” in the NT environment.

Performance on Mainframe Computers

Consider the paging parameters on a mainframe, and remember that Oracle can exploit a very large working set.

Free memory in a VAX/VMS environment is actually memory that is not currently mapped to any operating system process. On a busy system, free memory is likely to contain a page that belongs 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 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.

18

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
- How to Detect Contention Problems
- How to Solve Contention Problems
  - Reducing Contention for Rollback Segments
  - Reducing Contention for Multithreaded Server Processes
  - Reducing Contention for Parallel Server Processes
  - Reducing Contention for Redo Log Buffer Latches
  - Reducing Contention for the LRU Latch
  - Reducing Free List Contention

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, 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. Since 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 SPIN-COUNT 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: *Oracle8 Administrator's Guide* to understand which resources are used by various Oracle8 features.

How to Detect 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.

How to Solve Contention Problems

The rest of this chapter examines various kinds of contention and explains how to resolve problems. Contention may be for rollback segments, multithreaded server processes, parallel server processes, 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 reducing 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 is greater than 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 18–1 shows some general guidelines for choosing how many rollback segments to allocate based on the number of concurrent transactions on your database. These guidelines are appropriate for most application mixes.

Table 18–1 Choosing the Number of Rollback Segments

Number of Current Transactions (*n*)	Number of Rollback Segments Recommended
n < 16	4
16 <= n < 32	8
32 <= n	n / 4

Reducing Contention for Multithreaded Server Processes

In this section, you will learn how to reduce contention for some of the processes used by the Oracle's multithreaded server architecture:

- Reducing Contention for Dispatcher Processes
- Reducing Contention for Shared Server Processes

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 can be reflected by either of these symptoms:

- high busy rates for existing dispatcher processes
- steady increase in waiting time for responses in the response queues of existing dispatcher processes

Examining Busy Rates for Dispatcher Processes V$DISPATCHER contains statistics reflecting the activity of 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 reflect busy rates for dispatcher processes:

IDLE	the idle time for the dispatcher process in hundredths of a second
BUSY	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; rather, you must check statistics relevant to the workload you are applying. Thus, if the dispatcher processes for a specific protocol are busy more than 50% of the effective work time, then by adding dispatcher processes you may be able to 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

To add dispatcher processes while Oracle is running, use the MTS_DISPATCHERS parameter of the ALTER SYSTEM command.

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: *Oracle8 Administrator's Guide* for more information on adding dispatcher processes.

Enabling Connection Pooling

MTS_DISPATCHERS lets you enable various attributes for each dispatcher. Previously you could specify a protocol and an initial number of dispatchers. These attributes are specified in a position-dependent, comma-separated string assigned to MTS_DISPATCHERS. For example:

```
MTS_DISPATCHERS = "TCP, 3"
```

While remaining backwardly compatible with this format, Oracle8 supports a name-value syntax to let you specify existing and additional attributes in a position-independent case-insensitive manner. For example:

```
MTS_DISPATCHERS = "(PROTOCOL=TCP)(DISPATCHERS=3)"
```

One and only one of the following attributes is required: PROTOCOL, ADDRESS, or DESCRIPTION. Additional attributes are optional.

Note that the optional attribute POOL (or POO) is used to enable the Net8 connection pooling feature.

See Also: *Oracle8 SQL Reference* and the *Oracle Net8 Administrator's Guide* for more information about MTS_DISPATCHER specification and connection pooling.

Reducing Contention for Shared Server Processes

This section discusses how to identify contention for shared server processes and how to increase the maximum number of shared server processes.

Identifying Contention for Shared Server Processes

Contention for shared server processes can be reflected by a steady increase in waiting time for requests in the request queue. The dynamic performance table V$QUEUE contains statistics reflecting the request queue activity for shared server 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 requests in the queue:

WAIT	the total waiting time, in hundredths of a second, for all requests that have ever been in the queue
TOTALQ	the total number of requests that have ever been in the queue

Monitor these statistics occasionally while your application is running:

```
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 total wait time for all requests and total number of requests for the request queue. The result of this query might look like this:

```
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 it is processed.

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
```

Adding Shared Server Processes

Oracle automatically adds shared server processes if the load on existing processes increases drastically. Therefore, you are unlikely to improve performance simply by explicitly adding more shared server processes. However, if the number of shared server processes has reached the limit established by the initialization parameter MTS_MAX_SERVERS and the average wait time in the requests queue is still increasing, you may be able to improve performance by increasing the MTS_MAX_SERVERS value. The default value of this parameter is 20 and the maximum value varies depending on your operating system. You can then either allow Oracle to add shared server processes automatically, or explicitly add shared processes through one of these means:

- the MTS_SERVERS initialization parameter
- the MTS_SERVERS parameter of the ALTER SYSTEM command

See Also: *Oracle8 Administrator's Guide* for more information on adding shared server processes.

Reducing Contention for Parallel Server Processes

This section describes how to detect and alleviate contention for parallel server processes when using parallel execution:

- Identifying Contention for Parallel Server Processes
- Reducing Contention for Parallel Server Processes

Identifying Contention for Parallel Server Processes

Statistics in the V$PQ_SYSSTAT view are useful for determining the appropriate number of parallel server processes 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 server processes 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 parameter PARALLEL_MIN_SERVERS.

For example, if you have determined that the maximum number of concurrent parallel server processes that your machine can manage is 100, you should set PARALLEL_MAX_SERVERS to 100. Next, determine how many parallel server processes 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 server processes 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 server processes 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 Server Processes

If you find that typically there are fewer than PARALLEL_MIN_SERVERS busy at any given time, your idle parallel server processes 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 server processes 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, user processes can then copy new entries over the entries 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 table V$SYSSTAT. By default, this table 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 0. 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 or log switching. 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 latch: 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. Since 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.

The maximum size of a redo entry that can be copied on the redo allocation latch is specified by the initialization parameter LOG_SMALL_ENTRY_MAX_SIZE. The value of this parameter is expressed in bytes. The minimum, maximum, and default values vary depending on your operating system.

Redo Copy Latches

The user process first obtains the copy latch. 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 copy entries to the redo log buffer concurrently. The number of redo copy latches is determined by the parameter LOG_SIMULTANEOUS_COPIES; its default value is the number of CPUs available to your Oracle instance.

On single-CPU computers, there should be no redo copy latches, because only one process can by 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 table 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 table 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, since one request for the latch (the third request) was successful.
- The MISSES value increases by one, since the initial request for the latch resulted in waiting.
- The SLEEPS value increases by two, since 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 allo...     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 be 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 table V$WAITSTAT.

The V$WAITSTAT table contains block contention statistics. 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.

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 want to use one of these means instead:

- 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.

Part V

Optimizing Parallel Execution

Part V discusses parallel execution tuning. The chapters in Part 5 are:

- Chapter 19, “Tuning Parallel Execution”
- Chapter 20, “Understanding Parallel Execution Performance Issues”
- Chapter 21, “Diagnosing Parallel Execution Performance Problems”

19

Tuning Parallel Execution

Parallel execution can dramatically reduce response time for data-intensive operations on very large databases. This chapter explains how to tune your system for optimal performance of parallel operations.

- Introduction to Parallel Execution Tuning
- Step 1: Tuning System Parameters for Parallel Execution
- Step 2: Tuning Physical Database Layout for Parallel Execution
- Step 3: Analyzing Data

See Also: *Oracle8 Concepts*, for basic principles of parallel execution.

See your operating system-specific Oracle documentation for more information about tuning while using parallel execution.

Introduction to Parallel Execution Tuning

Parallel execution is useful for operations that access a large amount of data by way of large table scans, large joins, the creation of large indexes, partitioned index scans; bulk inserts, updates, deletes; aggregation or copying. It benefits systems with *all* of the following characteristics:

- symmetric multiprocessors (SMP), clusters, or massively parallel systems
- high I/O bandwidth (that is, many datafiles on many different disk drives)
- underutilized 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 any one of these conditions is *not* true for your system, parallel execution *may not* significantly help performance. In fact, on over-utilized systems or systems with small I/O bandwidth, parallel execution can impede system performance.

Note: In this chapter the term “parallel server process” designates a process (or a thread, on NT systems) that is performing a parallel operation, as distinguished from the product “Oracle Parallel Server”.

Step 1: Tuning System Parameters for Parallel Execution

Many initialization parameters affect parallel execution performance. For best results, start with an initialization file that is appropriate for the intended application.

Before starting the Oracle Server, set the initialization parameters described in this section. The recommended settings are guidelines for a large data warehouse (more than 100 gigabytes) on a typical high-end shared memory multiprocessor with one or two gigabytes of memory. Each section explains how to modify these settings for other configurations. Note that you can change some of these parameters dynamically with ALTER SYSTEM or ALTER SESSION statements. The parameters are grouped as follows:

- Parameters Affecting Resource Consumption for All Parallel Operations
- Parameters Affecting Resource Consumption for Parallel DML & Parallel DDL
- Parameters Enabling New Features
- Parameters Related to I/O

Parameters Affecting Resource Consumption for All Parallel Operations

The parameters discussed in this section affect the consumption of memory and other resources for all parallel operations, and in particular for parallel query. Chapter 20, “Understanding Parallel Execution Performance Issues” describes in detail how these parameters interrelate.

You must configure memory at two levels:

- at the Oracle level, so that 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 which control the total amount of virtual memory available, summed across all processes.

The SGA is typically part of the real physical memory. The SGA is static, of fixed size; if you want to change its size you must shut down the database, make the change, and restart the database.

The memory used in data warehousing operations is much more dynamic. It comes out of process memory: and both the size of a process’ memory and the number of processes can vary greatly.

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 less the SGA size. If you make it many times more than real memory, the paging rate may go up when the machine is overloaded at peak times.

As a general guideline for memory sizing, note that each process needs address space big enough for its hash joins. A dominant factor in heavyweight data warehousing operations is the relationship between memory, number of processes, and 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. Bear in mind, however, that sort performance degrades with increased memory use.

HASH_AREA_SIZE

Recommended value: Hash area size should be approximately half of the square root of *S*, where *S* is the size (in MB) of the smaller of the inputs to the join operation. (The value 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 the hash area might be 2MB, summed over all the parallel processes. Thus if you have 2 parallel processes, a minimum appropriate size might be 1MB hash area size. A smaller hash area would not be 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 that performs 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 could, however, affect performance.

Note that HASH_AREA_SIZE is relevant to parallel query operations, and to the query portion of DML or DDL statements.

See Also: "SORT_AREA_SIZE" on page 19-12

OPTIMIZER_PERCENT_PARALLEL

Recommended value: 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 that have low response time because of parallel execution, even if total resource used is not minimized.

The default value of OPTIMIZER_PERCENT_PARALLEL is 0, which parallelizes the plan that uses the least resource, if possible. Here, the execution time of the operation may be long because only a small amount of resource is used. A value of 100 causes the optimizer always to choose a parallel plan unless a serial plan would complete faster.

Note: Given an appropriate index a single record can be selected very quickly from a table, and does not require parallelism. A full scan to find the single row can be executed in parallel. Normally, however, each parallel process examines many rows. In this case response time of a parallel plan will be higher 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 resource is used. The parallel plan could use up to *D* times more resource, where *D* is 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.

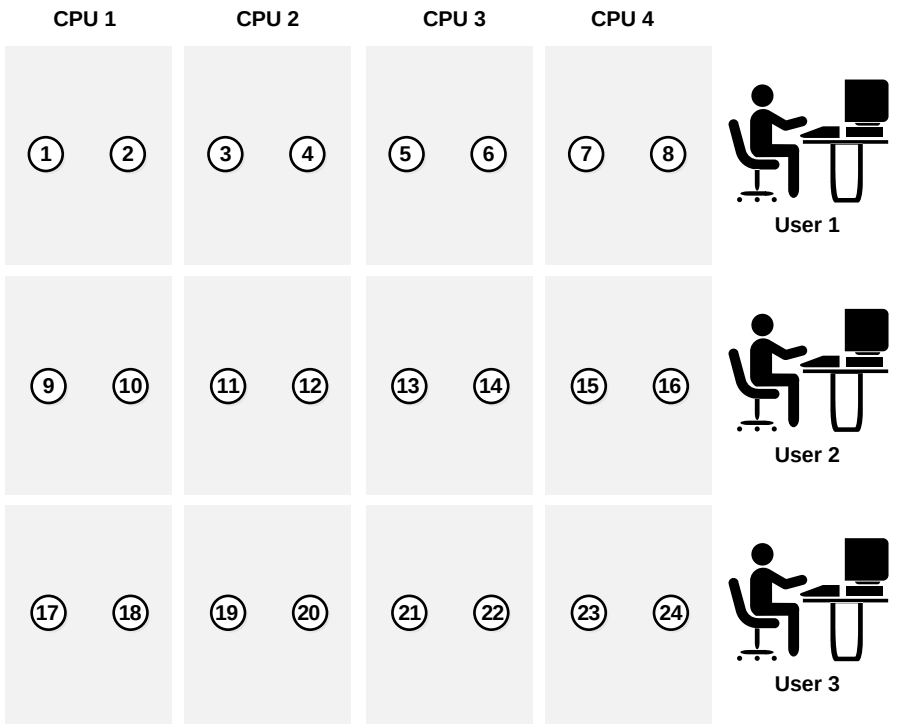
A nonzero setting of OPTIMIZER_PERCENT_PARALLEL will be overridden if you use a FIRST_ROWS hint or set OPTIMIZER_MODE to FIRST_ROWS.

PARALLEL_MAX_SERVERS

Recommended value: 2 * *CPUs* * *number_of_concurrent_users*

Most parallel operations need at most twice the number of parallel server processes (sometimes called “query servers”) as the maximum degree of parallelism attributed to any table in the operation. By default this is at most twice the number of CPUs. The following figure illustrates how the recommended value is derived.

Figure 19–1 PARALLEL_MAX_SERVERS = 2 * CPUs * Users



To support concurrent users, add more parallel 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.

Note that if a database’s users start up too many concurrent operations, Oracle may run out of parallel server processes. In this case, Oracle executes the operation sequentially or gives an error if PARALLEL_MIN_PERCENT is set.

When concurrent users use too many parallel server processes, memory contention (paging), I/O contention, or excessive context switching can occur. This contention could reduce system throughput to a level lower than if no parallel execution were 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 parallel server processes may restrict the number of concurrent users that can execute parallel operations, but system throughput will tend to remain stable.

To increase the number of concurrent users, you could restrict the number of concurrent sessions that various classes of user can have. For example:

- You can set a large limit for users running batch jobs.
- You could set a medium limit for users performing analyses.
- You could prohibit a particular class of user from using parallelism at all.

You can limit the amount of parallelism available to a given user by setting up a resource profile associated with the user. In this way you can limit the number of sessions or concurrent logons, which limits the number of parallel processes the user can have. (Each parallel server process working on your parallel execution statement is logged on as you—it counts against your limit of concurrent sessions.) For example, to limit a user to 10 processes, the DBA would set the user's limit to 11: one process for the parallel coordinator, and ten more parallel processes which would consist of two server sets. The user's maximum degree of parallelism would thus be 5.

On Oracle Parallel Server, if you have reached the limit of PARALLEL_MAX_SERVERS on an instance and you attempt to query a GV$ view, one additional parallel server process will be spawned for this purpose. The extra process is not available for any parallel operation other than GV$ queries.

See Also: "The Formula for Memory, Users, and Parallel Server Processes" on page 20-2 for further information on balancing concurrent users, degree of parallelism, and resources consumed.
Oracle8 Administrator's Guide for more information about managing resources with user profiles.
Oracle8 Parallel Server Concepts & Administration for more information on querying GV$ views.

PARALLEL_MIN_SERVERS

Recommended value: PARALLEL_MAX_SERVERS

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. Sometimes you may not be able to increase the maximum number of parallel server processes for an instance, because the maximum number depends on the capacity of the CPUs and the I/O bandwidth (platform-specific issues). However, if servers are continuously starting and shutting down, you should consider increasing the value of the parameter PARALLEL_MIN_SERVERS.

For example, if you have determined that the maximum number of concurrent parallel server processes that your machine can manage is 100, you should set PARALLEL_MAX_SERVERS to 100. Next determine how many parallel server processes the average operation needs, and how many 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. At any given time, there could be 80 parallel server processes busy on an instance. You should therefore set the parameter PARALLEL_MIN_SERVERS to 80.

Consider decreasing PARALLEL_MIN_SERVERS if fewer parallel server processes than this value are typically busy at any given time. Idle parallel server processes constitute unnecessary system overhead.

Consider increasing PARALLEL_MIN_SERVERS if more parallel server processes than this value are typically active, and the “Servers Started” statistic of V$PQ_SYSSTAT is continuously growing.

The advantage of starting these processes at startup is the reduction of process creation overhead. Note that Oracle reserves memory from the shared pool for these processes; you should therefore add additional memory using the initialization parameter SHARED_POOL_SIZE to compensate. Use the following formula to determine how much memory to add:

(CPUs + 2) * (PARALLEL_MIN_SERVERS) * 1.5 * (BLOCK_SIZE)

PARALLEL_ADAPTIVE_MULTI_USER

Recommended value: FALSE

On sites that have no clear usage profile, no consistent pattern of usage, you can use the PARALLEL_ADAPTIVE_MULTI_USER parameter to tune parallel execution for a multi-user environment. When set to TRUE, this parameter automatically reduces the requested degree of parallelism based on the current number of active parallel execution users on the system. The effective degree of parallelism is based on the degree of parallelism set by the table attributes or hint, divided by the total number of parallel execution users. Oracle assumes that the degree of parallelism provided has been tuned for optimal performance in a single user environment.

Note: This approach may not be suited for the general tuning policies you have implemented at your site. For this reason you should test and understand its effects, given your normal workload, before deciding to use it.

Note in particular that the degree of parallelism is not dynamic, but adaptive. It works best during a steady state when the number of users remains fairly constant. When the number of users increases or decreases drastically, the machine may be over- or under-utilized. For best results, use a parallel degree which is slightly greater than the number of processors. Based on available memory, the system can absorb an extra load. Once the degree of parallelism is chosen, it is kept during the entire query.

Consider, for example, a 16 CPU machine with the default degree of parallelism set to 32. If one user issues a parallel query, that user gets a degree of 32, effectively utilizing all of the CPU and memory resources in the system. If two users issue parallel queries, each gets a degree of 16. As the number of users on the system increases, the degree of parallelism continues to be reduced until a maximum of 32 users are running with degree 1 parallelism.

This parameter works best when used in single-node symmetric multiprocessors (SMPs). However, it can be set to TRUE when using Oracle Parallel Server if all of the following conditions are true:

* All parallel execution users connect to the same node.

* Instance groups are not configured.

* Each node has more than one CPU.

In this case, Oracle attempts to reduce the instances first, then the degree. If any of the above conditions is not met, and the parameter is set to TRUE, the algorithm may reduce parallelism excessively, causing unnecessary idle time.

SHARED_POOL_SIZE

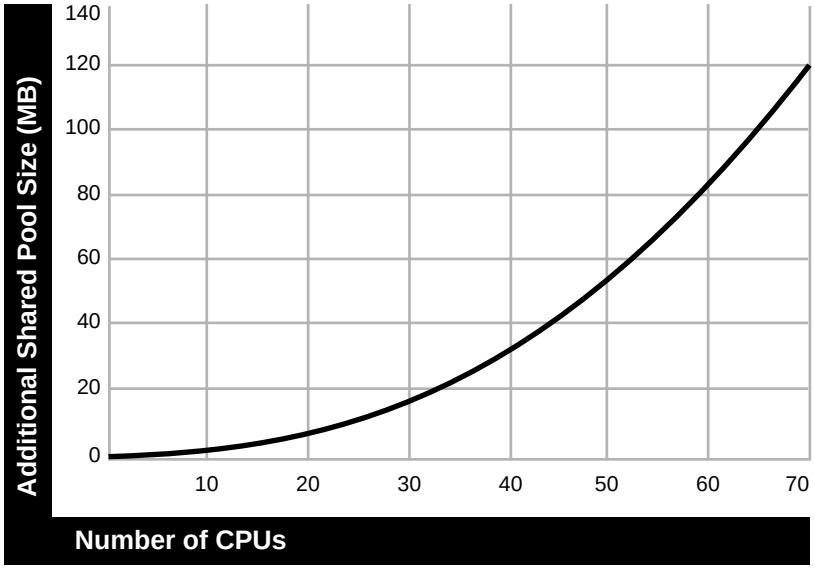
Recommended value: default plus
(3 * *msgbuffer_size)* * (*CPUs* + 2) * PARALLEL_MAX_SERVERS

Increase the initial value of this parameter to provide space for a pool of message buffers that parallel server processes can use to communicate with each other.

Note: The message buffer size might be 2 K or 4 K, depending on the platform. Check your platform vendor's documentation for details.

As illustrated in the following figure, assuming 4 concurrent users and 2 K buffer size, you would increase SHARED_POOL_SIZE by 6 K * (CPUs + 2) * PARALLEL_MAX_SERVERS for a pool of message buffers that parallel server processes use to communicate. This value grows quadratically with the degree of parallelism, if you set PARALLEL_MAX_SERVERS to the recommended value. (This is because the recommended values of PARALLEL_MAX_SERVERS and SHARED_POOL_SIZE both are calculated using the square root of the number of CPUs—a quadratic function.)

Figure 19–2 Increasing SHARED_POOL_SIZE with Degree of Parallelism



Parallel plans take up about twice as much space in the SQL area as serial plans, but additional space allocation is probably not necessary because generally they are not shared.

On Oracle Parallel Server, multiple CPUs can exist in a single node, and parallel operation can be performed across nodes. Whereas symmetric multiprocessor (SMP) systems use 3 buffers for connection, 4 buffers are used to connect between instances on Oracle Parallel Server. Thus you should normally have 4 buffers in shared memory: 2 in the local shared pool and 2 in the remote shared pool. The formula for increasing the value of SHARED_POOL_SIZE on Oracle Parallel Server becomes:

(4 * *msgbuffer_size)* * (*(CPUs_per_node * #nodes)* + 2) * (PARALLEL_MAX_SERVERS * *#nodes*)

Note that the degree of parallelism on Oracle Parallel Server is expressed by the number of CPUs per node multiplied by the number of nodes.

SORT_AREA_SIZE

Sample Range: 256 K to 4 M

This parameter specifies the amount of memory to allocate per parallel server process for sort operations. If memory is abundant on your system, you can benefit from setting SORT_AREA_SIZE to a large value. This can dramatically increase the performance of hash operations, because the entire operation 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 sorts and hashing operations. Instead, increase the size of the buffer cache so that data blocks from temporary sort segments can be cached in the buffer cache.

If the sort area is too small, an excessive amount of I/O will be required to merge a large number of runs. If the sort area size is smaller than the amount of data to sort, then the sort will spill 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 the 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 fast, because each parallel server can allocate this amount of memory for each sort. Monitor the operating system paging rate to see if too much memory is being requested.

Note that SORT_AREA_SIZE is relevant to parallel query operations and to the query portion of DML or DDL statements. All CREATE INDEX statements must do some sorting to generate the index. These include:

- CREATE INDEX
- direct-load INSERT (if an index is involved)
- ALTER INDEX ... REBUILD

See Also: "HASH_AREA_SIZE" on page 19-4

Parameters Affecting Resource Consumption for Parallel DML & Parallel DDL

Parallel INSERT, UPDATE, and DELETE require more resources than do serial DML operations. 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. Note that these parameters do *not* affect resources for queries.

See Also: *Oracle8 SQL Reference* for complete information about parameters.

TRANSACTIONS

For parallel DML, each parallel 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 degree of parallelism. You may thus need to increase the value of the TRANSACTIONS initialization parameter, which specifies the maximum number of concurrent transactions. (The default assumes no parallelism.) For example, if you have degree 20 parallelism 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.

ROLLBACK_SEGMENTS

The increased number of transactions for parallel DML necessitates many rollback segments. For example, one command with degree 5 parallelism uses 5 server transactions, which should be distributed among different rollback segments. The rollback segments should belong to tablespaces that have free space. The rollback segments should 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.

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 lots of 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 where the log buffers reside. In that case, stripe the log files across multiple disks in order to increase the I/O bandwidth.

Note that parallel DML generates a good deal more redo than does serial DML, especially during inserts, updates and deletes.

DML_LOCKS

This parameter specifies the maximum number of DML locks. Its value should equal the grand total of locks on 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 a equal amounts.

The following table shows the types of lock acquired by coordinator and server processes, for different types of parallel DML statements. Using this information you can figure the value required for these parameters. Note that a server process can work on one or more partitions, but a partition can only be worked on by one server process (this is different from parallel query).

Table 19–1 Locks Acquired by Parallel DML Statements

Type of statement	Coordinator acquires:	Each server process acquires:
Parallel UPDATE / DELETE into partitioned table; WHERE clause specifies the partition involved	1 table lock SX 1 partition lock X, per partition	1 table lock SX 1 partition lock NULL per partition 1 partition-wait lock X per partition
Parallel UPDATE / DELETE / INSERT into partitioned table	1 table lock SX partition locks X for all partitions	1 table lock SX 1 partition lock NULL per partition 1 partition-wait lock X per partition
Parallel INSERT into non-partitioned 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 parallel degree 100, assuming all partitions are involved in the parallel UPDATE/DELETE statement.

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 X

ENQUEUE_RESOURCES

This parameter sets the number of resources that can be locked by the distributed lock manager. Parallel DML operations require many more resources than serial DML. Therefore, you should increase the value of the ENQUEUE_RESOURCES and DML_LOCKS parameters, by equal amounts.

See Also: "DML_LOCKS" on page 19-14

Parameters Enabling New Features

Set these parameters in order to use the latest available Oracle 8 functionality.

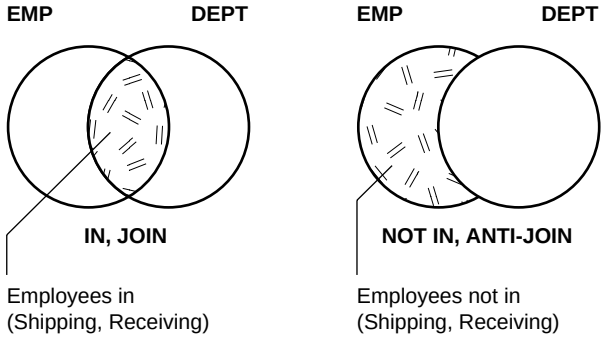
Note: Use partitioned tables instead of partition views. Partition views will be obsoleted in a future release.

ALWAYS_ANTI_JOIN

Recommended value: HASH

When set to HASH, this parameter causes the NOT IN operator to be evaluated in parallel using a parallel hash anti-join. Without this parameter set to HASH, NOT IN is evaluated as a (sequential) correlated subquery.

Figure 19–3 Parallel Hash Anti-join



As illustrated above, 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.

ALWAYS_SEMI_JOIN

Recommended value: default

When set to HASH, this parameter converts a correlated EXISTS subquery into a view query block and semi-join which is evaluated in parallel.

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 t2
           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 the following 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.

COMPATIBLE

Sample Value: 8.0.0

This parameter enables new features that may prevent you from falling back to an earlier release. To be sure that you are getting the full benefit of the latest performance features, set this parameter equal to the current release.

Note: Make a full backup before you change the value of this parameter.

PARALLEL_BROADCAST_ENABLE

Recommended value: default

You can 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 row sources of the small result set, such that a single table queue will send all of the small set's rows to each of the parallel servers which are processing the rows of the larger set. The result is enhanced performance.

Note that this parameter, which cannot be set dynamically, affects only hash joins and merge joins.

Parameters Related to I/O

Tune the following parameters to ensure that I/O operations are optimized for parallel execution.

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 14-26.

DB_BLOCK_SIZE

Recommended value: 8K or 16K

The database block size must be set when the database is created. If you are creating a new database, use a large block size.

DB_FILE_MULTIBLOCK_READ_COUNT

Recommended value: 8, for 8K block size; 4, for 16K block size

This parameter determines how many database blocks are read with a single operating system READ call. Many platforms limit the number of bytes read to 64K, limiting the effective maximum for an 8K block size to 8. Other platforms have a higher limit. For most applications, 64K is acceptable. In general, use the formula:

DB_FILE_MULTIBLOCK_READ_COUNT = 64K / DB_BLOCK_SIZE.

HASH_MULTIBLOCK_IO_COUNT

Recommended value: 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 join, consider increasing the value of HASH_MULTIBLOCK_IO_COUNT.

PARALLEL_EXECUTION_MESSAGE_SIZE

Recommended value: default

This parameter specifies the size of messages for parallel execution. The default value, which is operating system specific, should be adequate for most applications. Larger values would require a larger shared pool.

SORT_DIRECT_WRITES

Recommended value: AUTO

When this parameter is set to AUTO and SORT_AREA_SIZE is greater than 10 times the buffer size, this parameter causes the buffer cache to be bypassed for the writing of sort runs.

Reading through the buffer cache may result in greater path length, excessive memory bus utilization, and LRU latch contention on SMPs. Avoiding the buffer cache can thus provide performance improvement by a factor of 3 or more. It also removes the need to tune buffer cache and DBW*n* parameters.

Excessive paging is a symptom that the relationship of memory, users, and parallel server processes is out of balance. To rebalance it, you can reduce the sort or hash area size. You can limit the amount of memory for sorts if SORT_DIRECT_WRITES is set to AUTO but the SORT_AREA_SIZE is small. Then sort blocks will be cached in the buffer cache. Note that SORT_DIRECT_WRITES has no effect on hashing.

See Also: "HASH_AREA_SIZE" on page 19-4

SORT_READ_FAC

Recommended value: depends on disk speed

This value is a ratio that sets the amount of time needed to read a single database block, divided by the block transfer rate.

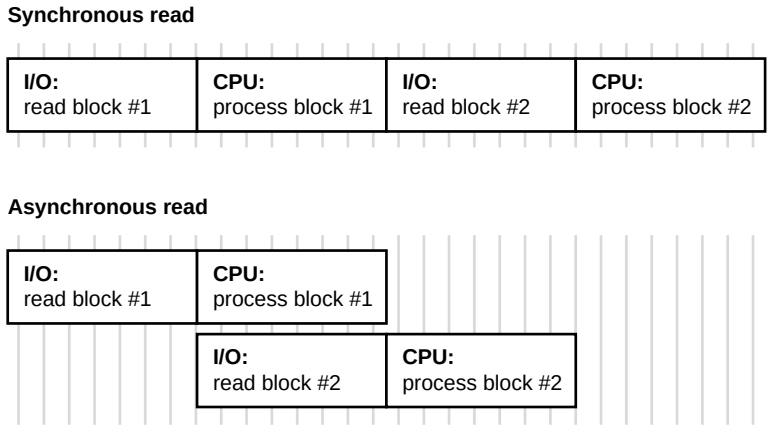
See Also: *Oracle8 Reference* and your Oracle platform-specific documentation for more information about setting this parameter.

DISK_ASYNCH_IO and TAPE_ASYNCH_IO

Recommended value: TRUE

These parameters enable or disable the operating system's asynchronous I/O facility. They allow parallel 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 19–4 Asynchronous Read



Asynchronous operations are currently supported with parallel table scans and hash joins only. They are not supported for sorts, or for serial table scans. In addition, this feature may require operating system specific configuration and may not be supported on all platforms. Check your Oracle platform-specific documentation.

Note: If asynchronous I/O behavior is not natively available, you can simulate it by deploying I/O server processes using the following parameters: DBWR_IO_SLAVES, LGWR_IO_SLAVES, BACKUP_DISK_IO_SLAVES, and BACKUP_TAPE_IO_SLAVES. Whether or not you use I/O servers is independent of the availability of asynchronous I/O from the platform. Although I/O server processes can be deployed even when asynchronous I/O is available, Oracle does not recommend this practice.

Step 2: Tuning Physical Database Layout 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 physical database layout you choose should depend 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. Parallel server processes execute the operation one granule at a time. The number of granules and their size affect the degree of parallelism you can use, and how well the work is balanced across parallel server processes.

Block Range Granules

Block range granules are the basic unit of most parallel operations (exceptions include such operations as parallel DML and CREATE LOCAL INDEX). 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 the table. Because they are based on physical data addresses, you can size block range granules to allow better load balancing. Block range granules permit dynamic parallelism that does not depend on static preallocation of tables or indexes. On SMP systems granules are located on different devices in order to drive as many disks as possible. On many MPP systems, block range granules are preferentially assigned to parallel server processes that have physical proximity to the disks storing the granules. Block range granules are used with global striping.

When block range granules are used predominantly for parallel access to a table or index, administrative considerations such as recovery, or using partitions for deleting portions of data, may influence partition layout more than performance considerations. If parallel execution operations frequently take advantage of partition pruning, it is important that the set of partitions accessed by the query be striped over at least as many disks as the degree of parallelism.

See Also: For MPP systems, see your platform-specific documentation

Partition Granules

When partition granules are used, a parallel server process works on an entire partition 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 degree of parallelism possible might be limited, and that load might not be well balanced across parallel 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.

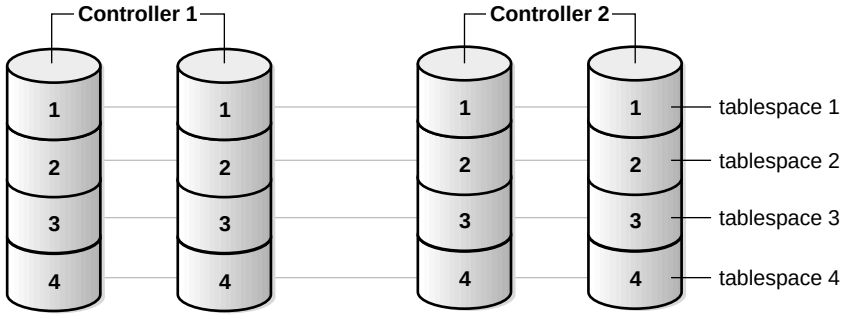
When partition granules are used for parallel access to a table or index, it is important that there be a relatively large number of partitions (at least three times the degree of parallelism), so that work can be balanced across the parallel server processes.

See Also: *Oracle8 Concepts* for information on disk striping and partitioning.

Striping Data

To avoid I/O bottlenecks, you should stripe all tablespaces accessed in parallel over at least as many disks as the degree of parallelism. *Stripe over at least as many devices as CPUs.* You should stripe tablespaces for tables, tablespaces for indexes, and temporary tablespaces. You must also spread the devices over controllers, I/O channels, and/or internal busses.

Figure 19–5 Striping Objects Over at Least as Many Devices as CPUs



To stripe data during load, use the FILE= clause of parallel loader to load data from multiple load sessions into different files in the tablespace. For any striping to be effective, you must 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.

Operating system striping with a large stripe size (at least 64K) is recommended, 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), which should enable this operation to reach the maximum I/O throughput that your platform can deliver. Note that 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 will be 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 that fewer disks are active at the same time. If you should encounter problems in this regard, increase the I/O size of scan operations (going, for example, from 64K to 128K), rather than changing the stripe size. Note that the maximum I/O size is platform specific (in a range, for example, of 64K 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 term 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 will automatically limit the contention. If availability is a major concern, associate this scheme with hardware redundancy (for example RAID5), which permits both performance and availability.

Manual Striping

Manual striping can be done on all platforms. This requires more DBA planning and effort to set up. For manual striping add multiple files, each on a separate disk, to each tablespace. The main problem with manual striping is that the degree of parallelism 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 being I/O bound. Also, this scheme is very sensitive to any skew in the datafile size which can affect the scalability of parallel scan operations.

See Also: *Oracle8 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.

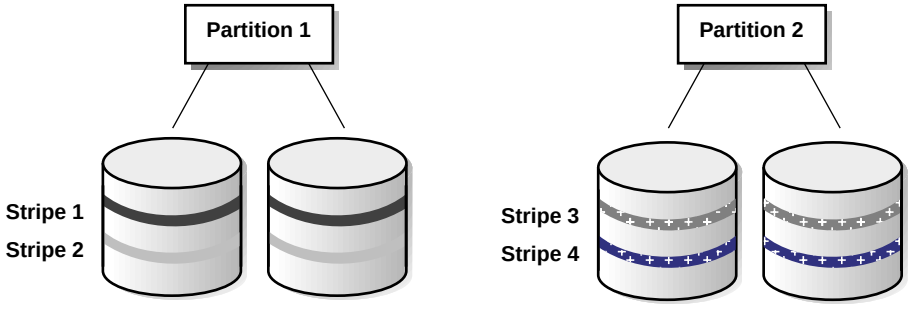
Local and Global Striping

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 Figure 19–6. There is no overlapping disk access, and no overlapping files.

Advantages of local striping are that if one disk fails it will not affect other partitions, and you still have some striping even if you have data in only one partition.

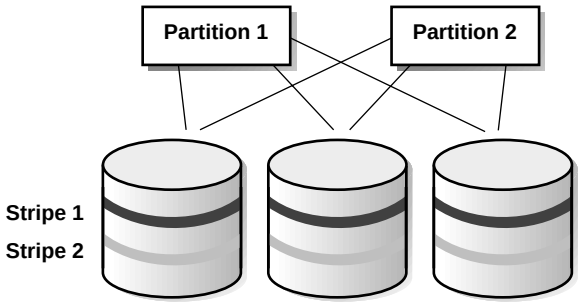
A disadvantage of local striping is that you need many more disks to implement it—each partition requires a few disks of its own. Another major disadvantage is that after partition pruning to only a single or a few partitions, the system will have limited I/O bandwidth. As a result, local striping is 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 19–6 Local Striping



Global striping, illustrated in Figure 19–7, entails overlapping disks and partitions.

Figure 19–7 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 query operations. A disadvantage of global striping is that if one disk fails, all partitions are affected.

See Also: "Striping and Media Recovery" on page 19-30

How to Analyze Striping

Relationships. To analyze striping, consider the following relationships:

Figure 19–8 Cardinality of Relationships

Figure 19–8 shows the cardinality of the relationships between objects in the storage system. For every table there may be *p* partitions; there may be *s* partitions for every tablespace, *f* files for every tablespace, and *m* files to *n* devices (a many-to-many relationship).

Goals. You may wish to stripe an object across devices for the sake of one of three goals:

- Goal 1: To optimize full table scans. This translates to placing a table on many devices.
- Goal 2: To optimize availability. This translates to restricting the tablespace to few devices.
- Goal 3: To optimize partition scans. This translates to 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 least intra-partition parallelism, place each partition in its own tablespace; do not used striped files; and use one file per tablespace. To minimize #2, set *f* and *n* equal to 1.

Notice that the solution that minimizes availability maximizes 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 if you are interested in 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:

$$\textit{Number of devices per table} = \frac{p \cdot f \cdot n}{s \cdot m}$$

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, one file, it should be striped over *n* devices.

1 x 1 x *n*

Maximum *t* partitions, every partition in its own tablespace, *f* files in each tablespace, each tablespace on 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.

$$\textit{Number of devices per tablespace} = \frac{f \cdot n}{m}$$

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.

$$\textit{Number of devices per partition} = \frac{f \cdot n}{s \cdot m}$$

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 that were stored on that disk. If all objects are striped over all disks, then loss of any disk takes down the entire database. Furthermore, you may need to restore all database files from backups, even if each file has only a small fraction actually 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 a device failure 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. Mirroring can be used effectively if you are able to reload read-only data from the original source tapes. If you do have a disk failure, restoring the data from the backup could involve lengthy downtime, whereas restoring it from a mirrored disk would enable your system to get back online quickly.

Even cheaper than mirroring is RAID technology, which avoids full duplication in favor of more expensive write operations. For read-mostly applications, this may suffice.

Note: RAID5 technology is particularly slow on write operations. This slowness may affect your database restore time to a point that RAID5 performance is unacceptable.

See Also: For a discussion of manually striping tables across datafiles, refer to "Striping Disks" on page 15-23.

For a discussion of media recovery issues, see "Backup and Recovery of the Data Warehouse" on page 6-8.

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

Partitioned tables and indexes can improve the performance of operations in a data warehouse. Partitioned tables and indexes allow at least the same parallelism as non-partitioned tables and indexes. In addition, partitions of a table can be pruned based on predicates and values in the partitioning column. Range scans on partitioned indexes can be parallelized, and insert, update and delete operations can be parallelized.

To avoid I/O bottlenecks when not all partitions are being scanned (because some have been eliminated), each partition should be spread over a number of devices. On MPP systems, those devices should be spread over multiple nodes.

Partitioned tables and indexes facilitate administrative operations by allowing them to operate on subsets of data. For example, a new partition can be added, an existing partition can be reorganized, or an old partition can be dropped with less than a second of interruption to a read-only application.

Consider using partitioned tables in a data warehouse when:

- very large tables are frequently scanned by a range predicate on a column that would make a good partitioning column (such as ORDER_DATE or PURCHASE_DATE)
- new data is loaded and old data is purged periodically and this can be translated into an add/drop of partitions
- administrative operations on large tables do not fit in the allotted batch window
- there is a need for parallel DML operations

Determining the Degree of Parallelism

If the data being accessed by a parallel operation (after partition pruning is applied) is spread over at least as many disks as the degree of parallelism, then most operations will be CPU-bound and a degree of parallelism ranging from the total number of CPUs to twice that number, is appropriate. Operations that tend to be I/O bandwidth bound can benefit from a higher degree of parallelism, especially if the data is spread over more disks. On sites with multiple users, you might consider using the PARALLEL_ADAPTIVE_MULTI_USER parameter to tune the requested degree of parallelism based on the current number of active parallel execution users. The following discussion is intended more for a single user environment.

Operations that tend to be I/O bandwidth bound are:

- selecting and counting all records in a table with a very simple or nonexistent WHERE clause
- nested loop joins using an index

Parallel operations that perform random I/O access (such as index maintenance of parallel update and delete operations) can saturate the I/O subsystem with a high number of I/Os, very much like an OLTP system with high concurrency. To ease this I/O problem, the data should be spread among more devices and disk controllers. Increasing the degree of parallelism will not help.

Oracle automatically computes the default parallel degree of a table as the minimum of the number of disks storing the table and the number of CPUs available. If, as recommended, you have striped objects over at least as many disks as you have CPUs, the default parallelism will always be the number of CPUs. Warehouse operations are typically CPU bound; thus the default is a good choice, especially if you are using the asynchronous readahead feature. However, because some operations are by nature synchronous (index probes, for example) an explicit setting of the parallel degree to twice the number of CPUs might be more appropriate. Consider reducing parallelism for objects that are frequently accessed by two or more concurrent parallel operations.

If you find that some operations are I/O bound with the default parallelism, and you have more disks than CPUs, you can override the usual parallelism with a hint that increases parallelism up to the number of disks, or until the CPUs become saturated.

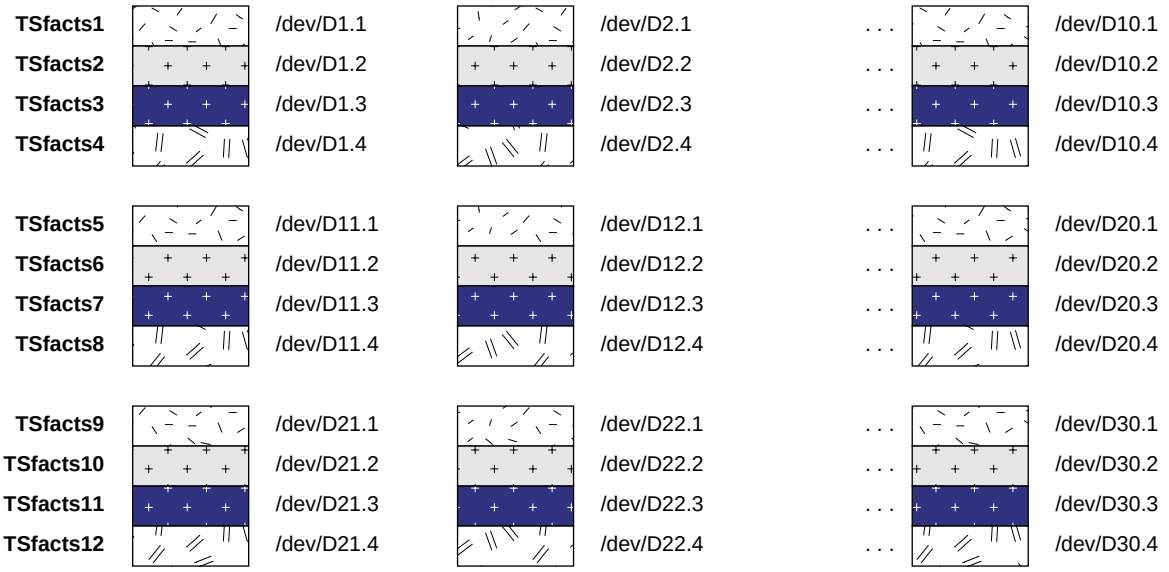
See Also: *Oracle8 Concepts*
"PARALLEL_ADAPTIVE_MULTI_USER" on page 19-9

Populating the Database Using Parallel Load

This section presents a case study which illustrates 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 G table is named FACTS.
- The system is a 10 CPU shared memory computer with more than 100 disk drives.
- Thirty disks (4 G 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 that a scan which accesses few partitions, or a single partition, can still 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 19–9.

Figure 19–9 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*

Note that 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 that you specify in the loader control file (such as, for example, 64K).

Tables or indexes can have an unlimited number of extents provided you have set the COMPATIBLE system parameter 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 the PERCENT_INCREASE parameter should be set to zero in order 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 20-12 for more information. Thus, each process should get an extent that will last for 3 to 5 minutes. Normally such an extent is at least 50MB for a large object. Too small an extent size will incur a lot of overhead, and this will affect 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 work nicely. Each partition will have 100 extents. The default storage parameters can be customized 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, which controls whether or not 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 that 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 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
```

Note that 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)
```

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 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 will partition 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 already partitioned outside Oracle.) 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 simultaneously to write to the same device.

Case 3

In Case 3 (illustrated in the example), the DBA wants precise control of 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. NFS mount of the data file on a remote node is not an optimal approach.

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 will be 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 of your 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 like 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: Oracle does that.

A disadvantage is that this approach requires all the partitions to be in the same tablespace; this minimizes availability.

Setting Up Temporary Tablespaces for Parallel Sort and Hash Join

For optimal space management performance you can use dedicated temporary tablespaces. As with the TSfacts tablespace, we first add a single datafile and later add the remainder in parallel.

```
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 to 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 120 GB example. With Oracle striping, simply remove the bad 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 1K, which is less than the extent size, the bad file will never be 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 tablespace. 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 degree of parallelism 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 degree of parallelism 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 back up or restore January data, you need only manage tablespace Tsidx1.

See Also: *Oracle8 Concepts* for a discussion of partitioned indexes.

Additional Considerations for Parallel DML Only

When parallel insert, update, or delete are to be performed on a data warehouse, some additional considerations are needed when designing the physical database. Note that these considerations do not affect parallel query operations. This section covers:

- 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

Limitation on the Degree of Parallelism

If you are performing parallel insert, update, or delete operations, the degree of parallelism will be equal to or les than the number of partitions in the table.

See Also: "Determining the Degree of Parallelism" on page 19-32

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 will only go 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 will be 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 num-

ber of transactions allocated within each data block) to a large value, such as the maximum degree of parallelism 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 (not to exceed 255).

If you run degree 10 parallelism 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 that all the server processes can make the change at the same time. If MAXTRANS is not large enough, the parallel DML operation will fail.

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 will leave 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 degree of parallelism is thus effectively limited by the smallest number of transaction free lists available on any of the global indexes which 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 degree of parallelism is limited to 30.

Note that 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 4K block has about 80 transaction free lists by default. The minimum number of transaction free lists is 25.

See Also: *Oracle8 Parallel Server Concepts & 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" in the V$SYSSTAT view, tune the DBW*n* process(es).

See Also: "Tuning the Redo Log Buffer" on page 14-7

[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 statement in Oracle8 Server, release 8.0. 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 is 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: *Oracle8 SQL Reference*

Step 3: Analyzing Data

After the data is loaded and indexed, analyze it. It is very important to analyze the data after any DDL changes or major DML changes. The ANALYZE command does not execute in parallel against a single table or partition. However, many different partitions of a partitioned table can be analyzed in parallel. Use the stored procedure DBMS_UTILITY.ANALYZE_PART_OBJECT to submit jobs to a job queue in order to analyze a partitioned table in parallel.

ANALYZE must be run twice in the following sequence to gather table and index statistics, and to create histograms. (The second ANALYZE statement creates histograms on columns.)

```
ANALYZE TABLE xyz ESTIMATE STATISTICS;
ANALYZE TABLE xyz ESTIMATE STATISTICS FOR ALL COLUMNS;
```

ANALYZE TABLE gathers statistics at the table level (for example, the number of rows and blocks), and for all dependent objects, such as columns and indexes. If run in the reverse sequence, the table-only ANALYZE will destroy the histograms built by the column-only ANALYZE.

An ESTIMATE of statistics produces accurate histograms for small tables only. Once the table is more than a few thousand rows, accurate histograms can only be ensured if more than the default number of rows is sampled. The sample size you choose depends on several factors: the objects to be analyzed (table, indexes, columns); the nature of the data (skewed or not); whether you are building histograms; as well as the performance you expect. The sample size for analyzing an index or a table need not be big; 1 percent is more than enough for tables containing more than 2000 rows.

Queries with many joins are quite sensitive to the accuracy of the statistics. Use the COMPUTE option of the ANALYZE command if possible (it may take quite some time and a large amount of temporary space). If you must use the ESTIMATE option, sample as large a percentage as possible (for example, 10%). If you use too high a sample size for ESTIMATE, however, this process may require practically the same execution time as COMPUTE. A good rule of thumb, in this case, is not to choose a percentage which causes you to access every block in the system. For example, if you have 20 blocks, and each block has 1000 rows, estimating 20% will cause you to touch every block. You may as well have computed the statistics!

Use histograms for data that is not uniformly distributed. Note that a great deal of data falls into this classification.

When you analyze a table, Oracle also analyzes all the different objects that are defined on that table: the table as well as its columns and indexes. Note that you

can use the ANALYZE INDEX statement to analyze the index separately. You may wish to do this when you add a new index to the table; it enables you to specify a different sample size. You can analyze all partitions of facts (including indexes) in parallel in one of two ways:

- Execute the following command. Make sure that job queues are enabled before the command is executed.

  ```
  EXECUTE DBMS_UTILITY.ANALYZE_PART_OBJECT(OBJECT_NAME=>'facts',
  COMMAND_TYPE=>'C');
  ```

- Concurrently, you can issue multiple ANALYZE commands concurrently.

It is worthwhile computing or estimating with a larger sample size the indexed columns, rather than the measure data. The measure data is not used as much: most of the predicates and critical optimizer information comes from the dimensions. A DBA should know which columns are the most frequently used in predicates.

For example, you might analyze the data in two passes. In the first pass you could obtain some statistics by analyzing 1% of the data. Run the following command to submit analysis commands to the job queues:

```
EXECUTE DBMS_UTILITY.ANALYZE_PART_OBJECT(OBJECT_NAME=>'facts',
COMMAND_TYPE=>'E',
SAMPLE_CLAUSE=>'SAMPLE 1 PERCENT');
```

In a second pass, you could refine statistics for the indexed columns and the index (but not the non-indexed columns):

```
EXECUTE DBMS_UTILITY.ANALYZE_PART_OBJECT(OBJECT_NAME=>'facts',
COMMAND_TYPE=>'C',
COMMAND_OPT=>'FOR ALL INDEXED
COLUMNS SIZE 1');
EXECUTE DBMS_UTILITY.ANALYZE_PART_OBJECT(OBJECT_NAME=>'facts',
COMMAND_TYPE=>'C',
COMMAND_OPT=>'FOR ALL INDEXES'):
```

The result will be a faster plan because you have collected accurate statistics on the most sensitive columns. You are spending more resources to get good statistics on high-value columns (indexes and join columns), and getting baseline statistics for the rest of the data.

Note: Cost-based optimization is always used with parallel execution and with partitioned tables. You must therefore analyze partitioned tables or tables used with parallel query.

20

Understanding Parallel Execution Performance Issues

This chapter provides a conceptual explanation of parallel execution performance issues, and additional performance techniques.

- Understanding Parallel Execution Performance Issues
- Parallel Execution Tuning Techniques

See Also: *Oracle8 Concepts*, for basic principles of parallel execution.

See your operating system-specific Oracle documentation for more information about tuning while using parallel execution.

Understanding Parallel Execution Performance Issues

- The Formula for Memory, Users, and Parallel Server Processes
- Setting Buffer Pool Size for Parallel Operations
- How to Balance the Formula
- Examples: Balancing Memory, Users, and Processes
- Parallel Execution Space Management Issues
- Optimizing Parallel Execution on Oracle Parallel Server

The Formula for Memory, Users, and Parallel Server Processes

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 server processes. The goal is to obtain the dramatic performance enhancement made possible by parallelizing certain operations, and by using hash joins rather than sort merge joins. This performance goal must often be balanced 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 20–1 defines high, medium, and low memory processes.

Analyze the maximum number of processes that can fit in memory as follows:

Figure 20–1 Formula for Memory/Users/Server Relationship

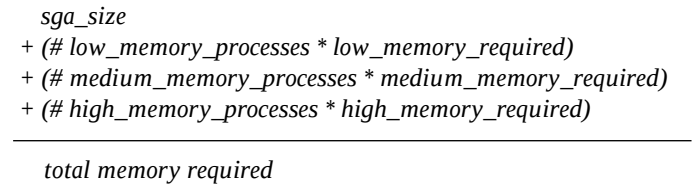


Table 20–1 Memory Requirements for Three Classes of Process

Class	Description
Low Memory Processes: 100K to 1MB	These 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); 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 multithreaded server, and support even more users.
Medium Memory Processes: 1MB to 10MB	This class of process includes large sorts; sort merge join; GROUP BY or ORDER BY operations returning a large number of rows; parallel insert operations which 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 do have a number of choices.
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 server process 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 parallel degree 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 and parallel DDL operations also depend upon the query portion of the statement.

Setting Buffer Pool Size for Parallel Operations

The formula whereby you can calculate the maximum number of processes your system can support (referred to here as *max_processes*) is:

$$\frac{\#\ low\_memory\_processes + \#\ medium\_memory\_processes + \#\ high\_memory\_processes}{max\_processes}$$

In general, if *max_processes* is much bigger than the number of users, you can consider running parallel operations. If *max_processes* is considerably less than the number of users, you must consider other alternatives, such as those described in “How to Balance the Formula” on page 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 the buffer pool can be made larger and thereby accommodate the inner table or index for a nested loop join.

See Also: *Oracle8 Concepts* for a comparison of hash joins and sort merge joins. "Tuning the Buffer Cache" on page 14-26 on setting buffer pool size.

How to Balance the Formula

Use the following techniques to balance the memory / users / server formula given in Figure 20–1:

- Oversubscribe, with Attention to Paging
- Reduce the Number of Memory-Intensive Processes
- Decrease Data Warehousing Memory per Process
- Decrease Parallelism for Multiple Users

Oversubscribe, 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 1G of memory, you might be able to support 1.2G of demand: the other 20% could be handled by the paging system.

You must, however, verify that a particular degree of oversubscription will be viable on your system 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 monitor paging activity in such a situation. If paging goes up dramatically, consider another alternative.

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 an I / O bound paging subsystem. Use your operating system monitor to check wait time: The sum of time waiting and time running equals 100%. If you are running close to 100% CPU, then you are not waiting. If you are waiting, it should not be on account of paging.

If wait time for paging devices exceeds 5%, it is a strong indication that you must 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.

Reduce the Number of Memory-Intensive Processes

Adjusting the Degree of Parallelism. You can adjust not only the number of operations that run in parallel, but also the degree of parallelism with which operations run. To do this, issue an ALTER TABLE statement with a PARALLEL clause, or use a hint. See the *Oracle8 SQL Reference* for more information.

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, and is easy to administer. More processes are then forced to run in serial mode.

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.

Decrease Data Warehousing Memory per Process

Note: 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 DSS 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 changing from hash join to merge join. You can use initialization parameters to limit available memory and thus force the optimizer to stay within certain bounds.

To do this, you can reduce HASH_AREA_SIZE to well below the recommended minimum (for example, to 1 or 2MB). Then you can let the optimizer choose sort merge join more often (as opposed to telling the optimizer never to use hash joins). In this way, hash join can still be used for small tables: the optimizer has a memory budget within which it can make decisions about which join method to use. Alternatively, you can use hints to force only certain queries (those whose response time is not critical) to use sort-merge joins rather than hash joins.

Remember that the recommended parameter values provide the best response time. If you severely limit these values you may see a significant effect on response time.

Moving Processes from High or Medium Memory Requirements to Low Memory Requirements. If you need to support thousands of users, you must create access paths such that operations do not touch much data.

- 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 rather than detailed data.
- Decrease the demand for ORDER BY sorts by creating indexes on frequently sorted columns.

Decrease Parallelism for Multiple Users

In general 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 2G of memory and a HASH_AREA_SIZE of 32MB can support about 60 parallel server processes. A 10 CPU machine can support up to 3 concurrent parallel operations (2 * 10 * 3 = 60). In order to support 12 concurrent parallel operations, you could override the default parallelism (reduce it); decrease HASH_AREA_SIZE; buy more memory, or use some combination of these three strategies. Thus you could ALTER TABLE *t* PARALLEL (DEGREE 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 taking such an approach is that when a single operation happens to be running, the system will use just half the CPU resource of the 10 CPU machine. The other half will be idle until another operation is started.

To determine whether your system is being fully utilized, you can 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 Processes

The examples in this section show how to evaluate the relationship between memory, users, and parallel server processes, and balance the formula given in Figure 20–1. They show concretely how you might adjust your system workload so as to accommodate the necessary number of processes and users.

Example 1

Assume that your system has 1G of memory, the degree of parallelism 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 (32MB) for best performance, then your system can support:

1 parallel operation (32MB * 10 * 2 = 640MB)
1 serial operation (32MB * 2 = 64MB)

This makes a total of 704MB. (Note that the memory is not significantly oversubscribed.)

Remember that every parallel, hash, or sort merge join operation takes a number of parallel server processes equal to twice the degree of parallelism (utilizing 2 server sets), and often each individual process of a parallel operation uses a lot of memory. Thus you can support many more users by having them run serially, or by having them run with less parallelism.

To service more users, you can drastically reduce hash area size to 2MB. You may then find that the optimizer switches some operations to sort merge join. 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.

Notice the trade-off above: 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 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 20–2. In this situation, you would have to make some choices. You could not allow everyone to run hash joins—even though they outperform sort merge joins—because you do not have the memory to support this level of workload.

You might consider it safe to oversubscribe at 50% because of the infrequent batch jobs during the day: 700MB * 1.5 = 1.05GB. This would give you enough virtual memory for the total workload.

Table 20–2 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 do hash joins that use a lot of memory.	You might take 20 parallel server processes, 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 big GROUP BY with join to produce a summary of data.) Twenty servers multiplied by 20MB equals 400MB of memory.
Analysts: interactive users who pull data into 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 do parallel operations because of memory requirements.) Ten such serial processes at 40MB apiece equals 400MB of memory.
Users: Several hundred users doing simple lookups of individual customer accounts, making reports on already joined, partially summarized data	To support hundreds of users doing low memory processes at about 0.5MB apiece, you might reserve 200MB.

Example 3

Suppose your system has 2G of memory, and you have 200 parallel server processes and 100 users doing heavy data warehousing operations involving hash joins. You decide to leave such tasks as index retrievals and small sorts out of the picture, concentrating 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 2G of memory might be used by the SGA, leaving 1.5G 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 20–2 Formula for Memory/User/Server Relationship: High-Memory Processes

$$high\_memory\_req'd = \frac{total\_memory}{\#\_high\text{-}memory\_processes} * 1.2 = \frac{1.5G * 1.2}{300} = \frac{1.8G}{300}$$

Here, 5MB = 1.8G / 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 force them to use other join methods in order 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 2G of memory and 10 users who want to run intensive data warehousing parallel operations concurrently and still have good performance. If you choose parallelism of degree 10, then the 10 users will require 200 processes. (Processes running big joins need twice the number of parallel server processes as the degree of parallelism, so you would set PARALLEL_MAX_SERVERS to 10 * 10 * 2.) In this example each process would get 1.8G / 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 2G of memory needs to support 1000 users, all of them running big operations, you must evaluate the situation carefully. Here, the per-user memory budget is only 1.8MB (that is, 1.8G 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 big 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 600K (1.8MB / 3). Such a small hash area size is likely to be ineffective, so you may opt to disable hash joins altogether.

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 doing big hash joins.
- Expect to support the 1000 users doing index lookups and joins that do not require large amounts of memory. Sort merge joins require less memory, but throughput will go down because they are not as efficient as 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 come into play when using parallel execution.

- ST (Space Transaction) Enqueue for Sorts and Temporary Data
- External Fragmentation

These issues become particularly important for parallel operation running on a parallel server, the more nodes involved, the more tuning becomes critical.

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 (more than 2 or 3 per minute) on the ST enqueue may result in poor scalability on Oracle Parallel Server systems with many nodes, or a timeout waiting for space management resources.

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 a parallel server. 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 40K for next extent size, because this will result in many requests for space per second.

If you are unable to allocate extents for various reasons, you can recoalesce the space by using the ALTER TABLESPACE ... COALESCE SPACE command. This should be done on a regular basis for temporary tablespaces in particular.

See Also: "Setting Up Temporary Tablespaces for Parallel Sort and Hash Join" on page 19-40

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 MINEXTENTS to the same value as NEXT, which should be equal to INITIAL; set PERCENT_INCREASE to zero. The system can handle this well with a few thousand extents per object, so you can set MAXEXTENTS to a few thousand. For tables that are not partitioned, the initial extent should be small.

Optimizing Parallel Execution on Oracle Parallel Server

This section describe several aspects of parallel execution on Oracle Parallel Server.

Lock Allocation

This section provides parallel execution tuning guidelines for optimal lock management on Oracle Parallel Server.

To optimize parallel execution on Oracle Parallel Server, you need to correctly set GC_FILES_TO_LOCKS. On Oracle Parallel Server 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 there will be no PCM locking at all.
- 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 will be 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 a million locks in order to read 1 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 might be comparable to the speed of hashed locking.

To speed up parallel DML operations, consider using hashed locking rather than DBA locking. A parallel server process 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. Since the parallel server process only works on non-overlapping files, there will be no lock pings.

The following guidelines impact 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 then never interfere with the rest of the database.

See Also: *Oracle8 Parallel Server Concepts & Administration* for a thorough discussion of PCM locks and locking parameters.

Allocation of Processes and Instances

Parallel execution assigns each instance a unique number, which is determined by the INSTANCE_NUMBER initialization parameter. The instance number regulates the order of instance startup.

Note: For Oracle Parallel Server, the PARALLEL_INSTANCE_GROUP parameter determines what instance group will be used for a particular operation. For more information, see *Oracle8 Parallel Server Concepts & Administration*.

Oracle computes a target degree of parallelism by examining the maximum of the degree for each table and other factors, before run time. At run time, a parallel operation is executed sequentially if insufficient parallel server processes are available. PARALLEL_MIN_PERCENT sets the minimum percentage of the target number of parallel server processes that must be available in order for the operation to run in parallel. When PARALLEL_MIN_PERCENT is set to *n*, an error message is sent if *n*

percent parallel server processes are not available. If no parallel server processes are available, a parallel operation is executed sequentially.

Load Balancing for Multiple Concurrent Parallel Operations

Load balancing is the distribution of parallel server processes to achieve even CPU and memory utilization, and to minimize remote I/O and communication between nodes.

When multiple concurrent operations are running on a single node, load balancing is done by the operating system. For example, if there are 10 CPUs and 5 parallel server processes, the operating system distributes the 5 processes among the CPUs. If a second node is added, the operating system still distributes the workload.

For a parallel server, however, no single operating system performs the load balancing: instead, parallel execution performs this function.

If an operation requests more than one instance, allocation priorities involve table caching and disk affinity.

Thus, if there are 5 parallel server processes, it is advantageous for them to run on as many nodes as possible.

In Oracle Server release 8.0, allocation of processes and instances is based on instance groups. With instance groups a parallel server system will be partitioned into disjoint logical subsystems. Parallel resources will be allocated out of a particular instance group only if the parallel coordinator is part of the group. This approach supports application and data partitioning.

See Also: *Oracle8 Parallel Server Concepts & Administration* for more information about instance groups.

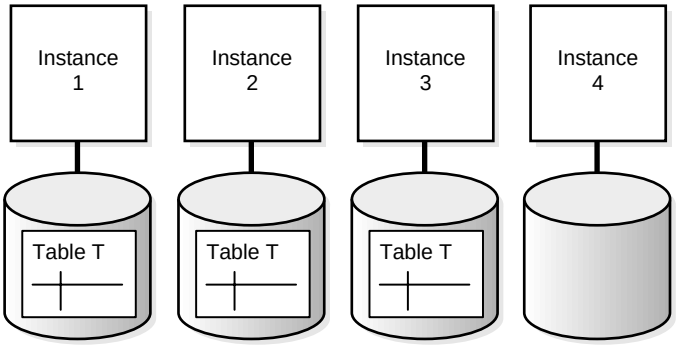
Disk Affinity

Some Oracle Parallel Server platforms use disk affinity. Without disk affinity, Oracle tries to balance the allocation evenly across instances; with disk affinity, Oracle tries to allocate parallel server processes 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. It can 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 20–3 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 will use instances 1, 2, and 3. Instance 4 will *not* be used. In contrast, without disk affinity instance 4 *would* be used.

Resource Timeout

A parallel DML transaction spanning Oracle Parallel Server instances may be waiting too long for a resource due to potential deadlock involving this transaction and other parallel or non-parallel DML transactions. Set the PARALLEL_TRANSACTION_RESOURCE_TIMEOUT parameter to specify how long a parallel DML transaction should wait for a resource before aborting.

See Also: *Oracle8 SQL Reference*

Parallel Execution Tuning Techniques

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
- Refreshing Tables in Parallel
- Using Hints with Cost Based Optimization
- Tuning Parallel Insert Performance

Overriding the Default Degree of Parallelism

The default degree of parallelism 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, you should consider increasing the default degree of parallelism. If it is memory bound, or several concurrent parallel operations are running, consider decreasing the default degree.

Oracle uses the default degree of parallelism 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 degree of parallelism that would be indicated by the number of CPUs, instances, and devices storing that table.

Use the following guidelines when adjusting the degree of parallelism:

- You can adjust the degree of parallelism either by using ALTER TABLE or by using hints.
- To increase the number of concurrent parallel operations, reduce the degree of parallelism.
- 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 degree of parallelism is 10, and this is I/O bound. You could first try parallel degree 12. If

still I/O bound, you could try parallel degree 24; if still I/O bound, you could try 36.

To override the default degree of parallelism:

1. Determine the maximum number of query servers your system can support.
2. Divide the parallel server processes among the estimated number of concurrent queries.

Rewriting SQL Statements

The most important issue for parallel query 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 20-22

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 query 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 huge 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 command 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 degree of parallelism. 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 degree of parallelism, parallel local index creation can be run with a higher degree of parallelism.

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 command is the only way in which you can specify the degree of parallelism for creating the index. If the degree of parallelism is not specified in the parallel clause of CREATE INDEX, then the number of CPUs is used as the degree of parallelism. If there is no parallel clause, index creation will be done serially.

Attention: 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 PARALLEL DEGREE 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 command 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 5;
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 query feature, see *Oracle8 Concepts*.
Refer to the *Oracle8 SQL Reference* for the complete syntax of the CREATE INDEX command.

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);
```

The CUSTOMER table can then be updated with the following SQL statement:

```
UPDATE /*+PARALLEL(customer,12)*/ customer
(SELECT 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
FROM customer, diff_customer
WHERE customer.c_key = diff_customer.c_key)
SET c_name = c_newname, c_addr = c_newaddr;
```

If the CUSTOMER table is partitioned, parallel DML can be used to further improve the response time. It could not be used with the original SQL statement because of the subquery in the SET clause.

See Also: "Rewriting SQL Statements" on page 20-18
Oracle8 Application Developer's Guide 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,12)*/ 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. Note that 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.

Attention: 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 give orders of magnitude improvements.

Note: If any table in a query has a parallel degree greater than one (including the default degree), 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 19-5. This parameter controls parallel awareness.

Tuning Parallel Insert Performance

This section provides an overview of parallel operation functionality.

- INSERT
- Direct-Load INSERT
- Parallelizing INSERT, UPDATE, and DELETE

See Also: *Oracle8 Concepts* for a detailed discussion of parallel Data Manipulation Language and degree of parallelism.
For a discussion of parallel DML affinity, please see *Oracle8 Parallel Server Concepts & Administration*.

INSERT

Oracle8 INSERT functionality can be summarized as follows:

Table 20–3 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 will be 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.

Note that the APPEND hint applies to both serial and parallel insert: even serial insert will be faster if you use it. 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: *Oracle8 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 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. Note, in general, that 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: *Oracle8 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.

Note also that 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,4) */ INTO emp
SELECT /*+ PARALLEL(acme_emp,4) */ *
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 parallel DML 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.

Parallel UPDATE and DELETE can be done only on partitioned tables.

Example 1

Give a 10% salary raise to all clerks in Dallas.

```
UPDATE /*+ PARALLEL(emp,5) */ 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, which will now be outsourced.

```
DELETE /*+ PARALLEL(emp,2) */ FROM emp
WHERE deptno IN
(SELECT deptno FROM dept WHERE dname='ACCOUNTING');
```

Again, the parallelism will be applied to the scan as well as update operation on table EMP.

Additional PDML Examples The following examples show the use of parallel DML.

Note: As these examples demonstrate, you must enable parallel DML before using the PARALLEL or APPEND hints. You must issue a COMMIT or ROLLBACK command immediately after executing parallel INSERT, UPDATE, or DELETE. You can issue no other SQL commands before committing or rolling back.

The following statement enables parallel DML:

```
ALTER SESSION ENABLE PARALLEL DML;
```

Serial as well as parallel direct-load insert requires commit or rollback immediately afterwards.

```
INSERT   /*+ APPEND NOPARALLEL(table1) */ INTO table1
```

A select statement issued at this point would fail, with an error message, because no SQL can be performed before a COMMIT or ROLLBACK is issued.

```
ROLLBACK;
```

After this ROLLBACK, a SELECT statement will succeed:

```
SELECT * FROM V$PQ_SESSTAT;
```

Parallel update likewise requires commit or rollback immediately afterwards:

```
UPDATE   /*+ PARALLEL(table1,2) */ table1
SET col1 = col1 + 1;
COMMIT;
SELECT * FROM V$PQ_SESSTAT;
```

As does parallel delete:

```
DELETE   /*+ PARALLEL(table3,2) */ FROM table3
WHERE col2 < 5;
COMMIT;
SELECT * FROM V$PQ_SESSTAT;
```

21

Diagnosing Parallel Execution Performance Problems

This section summarizes common tools and techniques you can use to obtain performance feedback on parallel operations.

- Diagnosing Problems
- Executing Parallel SQL Statements
- Using EXPLAIN PLAN to See How an Operation Is Parallelized
- Using the Dynamic Performance Views
- Checking Operating System Statistics
- Minimum Recovery Time
- Parallel DML Restrictions

See Also: *Oracle8 Concepts*, for basic principles of parallel execution.

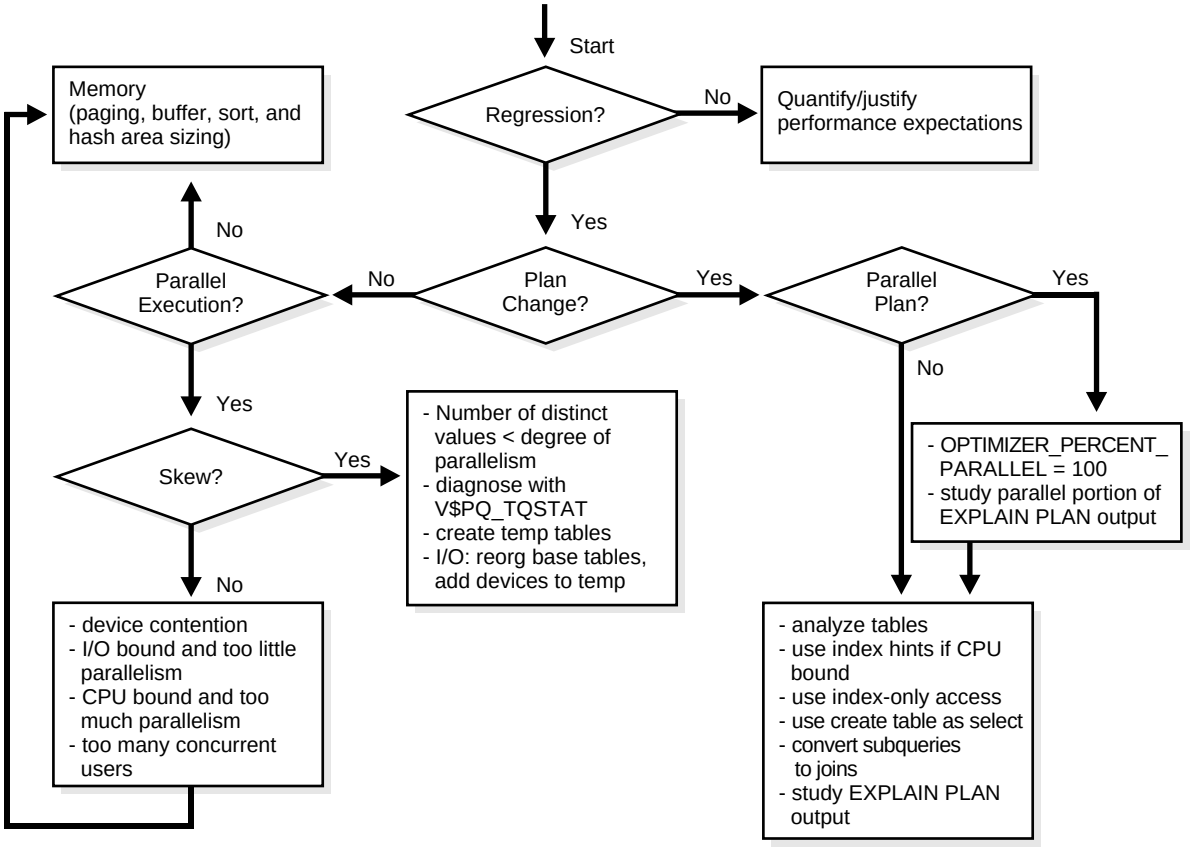
See your operating system-specific Oracle documentation for more information about tuning while using parallel execution.

Diagnosing Problems

Use the decision tree in Figure 21–1 to diagnose parallel performance problems. Some key issues are the following:

- Quantify your performance expectations, to determine whether there is a problem.
- Determine whether a problem pertains to optimization (such as an inefficient plan, which may require re-analyzing tables or adding hints) or to execution (such as simple operations like scanning, loading, grouping, or indexing running much slower than published guidelines).
- Determine whether the problem arises when running in parallel (such as load imbalance or resource bottleneck) or whether the problem is also present when running serially.

Figure 21–1 Parallel Execution Performance Checklist



Is There Regression?

Does parallel execution's actual performance deviate from what you expected? If performance is as you expected, can you justify the notion that there is a performance problem? Perhaps you have a desired outcome in mind, to which you are comparing the current outcome. Perhaps you have a justifiable performance expectation which the system is not achieving. You might have achieved this level of performance or particular execution plan in the past, but now, with a similar environment and operation, this is not being met.

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 in new data over the course of time, you can compare any new execution plan with the old plan. You can take this approach either proactively or reactively.

Alternatively, you may find that you get a plan that works better if you use hints. You may want to understand why hints were necessary, and figure out 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, look to see whether you had OPTIMIZER_PERCENT_PARALLEL set to 100%.

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:

- If you want a parallel plan, but the optimizer has not given you one, try increasing OPTIMIZER_PERCENT_PARALLEL to 100 and see if this improves performance.
- 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 gives better results.

See Also: Parallel EXPLAIN PLAN tags are defined in Table 23–2.

Is There a Serial Plan?

If the execution plan is (or should be) serial, consider the following strategies:

- Use an index. Sometimes adding the right 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 particularly important if you are performing many joins; 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.
- Note 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 the troublesome parts of the query by breaking it into steps and analyzing each step.

See Also: *Oracle8 Concepts* regarding CREATE TABLE ... AS SELECT
"Step 3: Analyzing Data" on page 19-45 regarding COMPUTE and ESTIMATE

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 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, you can look at the V$SESSTAT, V$PQ_SESSTAT and V$PQ_SYSSTAT views to see the number of server processes used, and other information for the session and system.

See Also: "Querying the Dynamic Performance Views: Example" on page 21-12

Is There Skew?

If parallel execution is occurring, is there unevenness in workload distribution (skew)? 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 server process. 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 which 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?

Executing Parallel SQL Statements

After analyzing your tables and indexes, you should be able to run operations and see speedup 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 See How an Operation Is Parallelized

Use an EXPLAIN PLAN statement to view the sequential and parallel operation plan. The OTHER_TAG column of the plan table summarizes how each plan step is parallelized, and describes the text of the operation used by parallel server processes for each operation. Optimizer cost information for selected plan steps is given in the COST, BYTES, and CARDINALITY columns. Table 23–2 summarizes the meaning of the OTHER_TAG column.

EXPLAIN PLAN thus provides detailed information as to how specific operations are being performed. You can then change the execution plan for better performance. For example, if many steps are serial (where OTHER_TAG is blank, serial to parallel, or parallel to serial), then the parallel controller could be a bottleneck. Consider the following SQL statement, which summarizes sales data by region:

```
EXPLAIN PLAN SET STATEMENT_ID = ‘Jan_Summary’ FOR
SELECT dim_1 SUM(meas1) FROM facts WHERE dim_2 < ‘02-01-1995’
GROUP BY dim_1
```

This script extracts a compact hierarchical plan from the EXPLAIN PLAN output:

```
SELECT
SUBSTR( LPAD(' ',2*(level-1)) ||
DECODE(id, 0, statement_id, operation) ||
' '  || options || ' ' || object_name ||
' [' || partition_start || ',' || partition_stop ||
'] ' || other_tag,
1, 79)  "step [start,stop] par"
FROM plan_table
START WITH id = 0
CONNECT BY PRIOR id = parent_id
AND PRIOR NVL(statement_id,' ') =
NVL(statement_id);
```

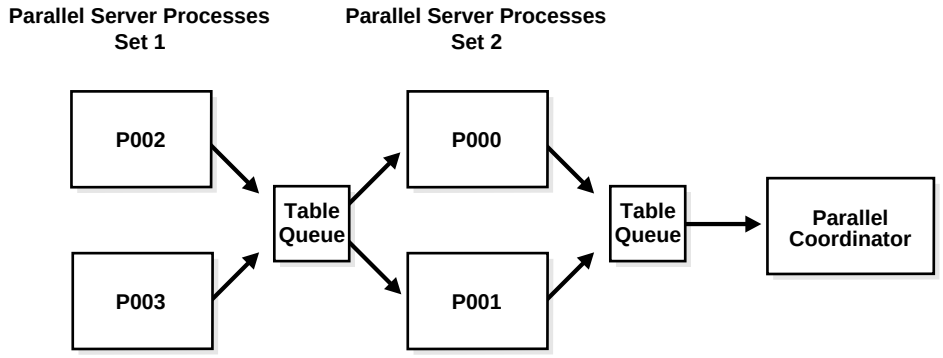
Following is the query plan for “Jan_Summary”:

```
Jan_Summary [,]
SORT GROUP BY [,] PARALLEL_TO_SERIAL
TABLE ACCESS FULL facts  [NUMBER(1),NUMBER(1)] PARALLEL_TO_PARALLEL;
```

Each parallel server scans a portion of the first partition of the facts table. All other partitions are pruned, as shown by the stop and start partition number.

Figure 21–2 illustrates how, with the PARALLEL_TO_PARALLEL keyword, data from the partitioned table is redistributed to a second set of parallel server processes for parallel grouping. Query server set 1 executes the table scan from the preceding example. The rows coming out are repartitioned through the table queue to parallel server set 2, which executes the GROUP BY operation. Because the GROUP BY operation indicates PARALLEL_TO_SERIAL, another table queue collects its results and sends it to the parallel coordinator, and then to the user.

Figure 21–2 Data Redistribution among Parallel Server Processes



As a rule, if the PARALLEL_TO_PARALLEL keyword exists, there will be two sets of parallel server processes. This means that for grouping, sort merge, or hash joins, twice the number of parallel server processes are assigned to the operation. This requires redistribution of data or rows from set 1 to set 2. If there is no PARALLEL_TO_PARALLEL keyword, then the operation gets just one set of servers. Such serial processes include aggregations, such as COUNT(*) FROM facts or SELECT * FROM facts WHERE DATE = '7/1/94'.

For non-distributed operations, the OBJECT_NODE column gives the name of the table queue. If the PARALLEL_TO_PARALLEL keyword exists, then the EXPLAIN PLAN of the parent operation should have SQL that references the child table queue in its FROM clause. In this way the plan describes the order in which the output from operations is consumed.

See Also: Chapter 23, "The EXPLAIN PLAN Command"

Using the Dynamic Performance Views

Dynamic performance views list internal Oracle8 data structures and statistics that you can query periodically to monitor progress of a long-running operation. When used in conjunction with data dictionary views, these tables provide a wealth of information. The challenge is visualizing the data and then acting upon it.

Note: On Oracle Parallel Server, global versions of these views aggregate the information over multiple instances. The global views have analogous names such as GV$FILESTAT for V$FILESTAT, and so on.

See Also: *Oracle8 Parallel Server Concepts & Administration* for more information about global dynamic performance views.

V$FILESTAT

This view sums read and write requests, number of blocks, and service times for every datafile in every tablespace. It can help you diagnose I/O problems and workload distribution problems.

The file numbers listed in V$FILESTAT can be joined to those in the DBA_DATA_FILES view to group I/O by tablespace or to find the filename for a given file number. By doing 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 correlating I/O activity to the EXPLAIN PLAN output is a good way to follow progress.

V$PARAMETER

This view lists the name, current value, and default value of all system parameters. In addition, the view indicates whether the parameter may be modified online with an ALTER SYSTEM or ALTER SESSION command.

V$PQ_SESSTAT

This view is valid only when queried from a session that is executing parallel SQL statements. Thus it cannot be used to monitor a long running operation. It gives summary statistics about the parallel statements executed in the session, including total number of messages exchanged between server processes and the actual number of parallel server processes used.

V$PQ_SLAVE

This view tallies the current and total CPU time and number of messages sent and received per parallel server process. It can be monitored during a long-running operation. Verify that there is little variance among processes in CPU usage and number of messages processed. A variance may indicate a load-balancing problem. Attempt to correlate the variance to a variance in the base data distribution. Extreme imbalance could indicate that the number of distinct values in a join column is much less than the degree of parallelism. See “Parallel CREATE TABLE ... AS SELECT” in *Oracle8 Concepts* for a possible workaround.

V$PQ_SYSSTAT

The V$PQ_SYSSTAT view aggregates session statistics from all parallel server processes. It sums the total parallel server message traffic, and gives the status of the pool of parallel server processes.

This view can help you determine the appropriate number of parallel server processes for an instance. The statistics that are particularly useful are “Servers Busy”, “Servers Idle”, “Servers Started”, and “Servers Shutdown”.

Periodically examine V$PQ_SYSSTAT to determine whether the parallel server processes for the instance are actually busy, as follows:

```
SELECT * FROM V$PQ_SYSSTAT
WHERE statistic = "Servers Busy";

STATISTIC            VALUE
-------------------- -----------
Servers Busy         70
```

V$PQ_TQSTAT

This view provides a detailed report of message traffic at the level of the table queue. It is valid only when queried from a session that is executing parallel SQL statements. A table queue is the pipeline between parallel server groups or between the parallel coordinator and a parallel server group or between a parallel server group and the coordinator. These table queues are represented in the plan by the tags PARALLEL_TO_PARALLEL, SERIAL_TO_PARALLEL, or PARALLEL_TO_SERIAL, respectively.

The view contains a row for each parallel server process that reads or writes each table queue. A table queue connecting 10 consumers to 10 producers will have 20 rows in the view. Sum the bytes column and group by TQ_ID (table queue identifier) for 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 imbalance. Drill down on large variances to determine whether the producers start out with unequal distributions of data, or whether the distribution itself is skewed. The latter may indicate a low number of distinct values.

For many of the dynamic performance views, the system parameter TIMED_STATISTICS must be set to TRUE in order to get the most useful information. You can use ALTER SYSTEM to turn TIMED_STATISTICS on and off dynamically.

See Also: Chapter 22, "The Dynamic Performance Views"

V$SESSTAT and V$SYSSTAT

The V$SESSTAT view provides statistics related to parallel execution 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.

Querying the Dynamic Performance Views: Example

The following example illustrates output from two of these views:

```
SQLDBA> update /*+ parallel (iemp, 2) */ iemp set empno = empno +1;
91 rows processed.
SQLDBA> commit;
Statement processed.
SQLDBA> select * from v$pq_sesstat;
STATISTIC                      LAST_QUERY SESSION_TOTAL
------------------------------ ---------- ----------
Queries Parallelized                    0          0
DML Parallelized                        1          2
DFO Trees                               1          2
Server Threads                          2          0
Allocation Height                       2          0
Allocation Width                        0          0
Local Msgs Sent                        34         60
Distr Msgs Sent                         0          0
Local Msgs Recv'd                      34         60
Distr Msgs Recv'd                       0          0
11 rows selected.
```

```
SQLDBA> select * from v$pq_sysstat;
STATISTIC                       VALUE
------------------------------ ----------
Servers Busy                            0
Servers Idle                            2
Servers Highwater                       2
Server Sessions                         4
Servers Started                         2
Servers Shutdown                        0
Servers Cleaned Up                      0
Queries Initiated                       0
DML Initiated                           2
DFO Trees                               2
Local Msgs Sent                        60
Distr Msgs Sent                         0
Local Msgs Recv'd                      60
Distr Msgs Recv'd                       0
15 rows selected.
```

In V$PQ_SESSTAT, some of the statistics provide the following information.

DML Parallelized	number of statements with insert, delete and update that were parallelized by the last operation and by this session
Queries Parallelized	number of all other parallel statements
DFO Trees	number of fragments of query plan that were parallelized
Server Threads	total number of server processes (typically 2x degree)
Allocation Height	requested number of servers on each instance
Allocation Width	requested number of instances

In V$PQ_SYSSTAT, the "DML Initiated" statistic indicates the number of DML operations done in the system.

Note that statements such as INSERT ... SELECT are treated as a single DML statement, not as one DML statement and one query.

See Also: *Oracle8 SQL Reference* for information about statistics.

Checking 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.

Minimum Recovery Time

If the system requires ten minutes to run an operation, it will take at least 10 minutes to roll back the operation: this is the best performance achievable. If there is some instance failure and some system failure, then recovery time will increase because not all the server processes are available to provide rollback.

Parallel DML Restrictions

You must either commit or roll back directly after you issue a parallel INSERT, UPDATE, or DELETE statement, or a serial insert with the APPEND hint. You can perform no other SQL commands until this is done.

Discrete transactions are not supported for parallel DML.

A session that is enabled for parallel DML may put transactions in the session in a special mode. If any DML statement in a transaction modifies a table in parallel, no subsequent serial or parallel query or DML statement can access the same table again in that transaction. This means that the results of parallel modifications cannot be seen during the transaction.

A complete listing of parallel DML and direct-load insert restrictions is found in *Oracle8 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.

Part VI

Performance Diagnostic Tools

Part VI discusses the tools available to diagnose system problems and tune system performance. The chapters in Part VI are:

- Chapter 22, “The Dynamic Performance Views”
- Chapter 23, “The EXPLAIN PLAN Command”
- Chapter 24, “The SQL Trace Facility and TKPROF”
- Chapter 25, “Using Oracle Trace”
- Chapter 26, “Registering Applications”

22

The Dynamic Performance Views

Dynamic performance views are useful for identifying instance-level performance problems. Whereas the underlying X$ tables represent internal data structures that can be modified by SQL statements, the V$ views allow users other than SYS read-only access to this data. This chapter describes the views of the greatest use for both performance tuning and ad hoc investigation—for example, when users report a sudden deterioration in response time.

- Instance-Level Views for Tuning
- Session-Level or Transient Views for Tuning
- Current Statistic Value and Rate of Change

See Also: For complete information on all dynamic performance tables, see *Oracle8 Reference*.

Instance-Level Views for Tuning

These views concern the instance as a whole and record statistics either since startup of the instance or (in the case of the SGA statistics) the current values, which will remain constant until altered by some need to reallocate SGA space. Cumulative statistics are from startup.

Table 22–1 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	Dynamic view. 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 22–2 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 further to see whether such waits occur frequently and whether they can be correlated with other phenomena, such as the use of particular modules.

Current Statistic Value and Rate 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:

```
SQL> @STAT GET
```

It is preferable, however, to use some mechanism that records the change in the statistic(s) over a known period of time.

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: This script must be run at least twice, because the first time it is run, it will initialize the SQL*Plus variables.

23

The EXPLAIN PLAN Command

This chapter shows how to use the SQL command EXPLAIN PLAN. It covers the following topics:

- Introduction
- Creating the Output Table
- Output Table Columns
- Formatting EXPLAIN PLAN Output
- EXPLAIN PLAN Restrictions

See Also: For the syntax of the EXPLAIN PLAN command, see the *Oracle8 SQL Reference.*

Introduction

The EXPLAIN PLAN command displays the execution plan chosen by the Oracle optimizer for SELECT, UPDATE, INSERT, and DELETE statements. A statement's execution plan is the sequence of operations that Oracle performs to execute the statement. By examining the execution plan, you can see exactly how Oracle executes your SQL statement.

EXPLAIN PLAN results alone cannot tell you which statements will perform well, and which badly. For example, just because EXPLAIN PLAN indicates that a statement will use an index does not mean that the statement will run quickly. The index might be very inefficient! Use EXPLAIN PLAN to determine the access plan and to test modifications to improve the performance.

It is not necessarily useful to subjectively evaluate the plan for a statement, and decide to tune it based only on the execution plan. Instead, you should examine the statement's *actual resource consumption*. For best results, use the Oracle Trace or SQL trace facility and TKPROF to examine performance information on individual SQL statements.

Attention: EXPLAIN PLAN tells you the execution plan the optimizer would choose if it were to produce an execution plan for a SQL statement at the current time, with the current set of initialization and session parameters. However, this plan is not necessarily the same as the plan that was used at the time the given statement was actually executed. The optimizer bases its analysis on many pieces of data—some of which may have changed! Furthermore, because the behavior of the optimizer is likely to evolve between releases of the Oracle Server, output from the EXPLAIN PLAN command will also evolve. Changes to both the optimizer and EXPLAIN PLAN output will be documented as they arise.

The row source count values appearing in EXPLAIN PLAN output identify the number of rows that have been processed by each step in the plan. This can help you to identify where the inefficiency in the query lies (that is, the row source with an access plan that is performing inefficient operations).

See also: Chapter 24, "The SQL Trace Facility and TKPROF"
Chapter 25, "Using Oracle Trace"

Creating the Output Table

Before you can issue an EXPLAIN PLAN statement, you must 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 command 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);
```

Output Table Columns

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

Table 23–1 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.
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
OPTIONS	A variation on the operation described in the OPERATION column.
OBJECT_NODE	The name of the database link used to reference the object (a table name or view name). For local queries using the parallel query option, 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. Note that view expansion will result in unpredictable numbers.

Table 23–1 PLAN_TABLE Columns

OBJECT_TYPE	A modifier that provides descriptive information about the object; for example, NON-UNIQUE for indexes.
OPTIMIZER	The current mode of the optimizer.
SEARCH_COLUMNS	Not currently used.
ID	A number assigned to each step in the execution plan.
PARENT_ID	The ID of the next execution step that operates on the output of the ID step.
POSITION	The order of processing for steps that all have the same PARENT_ID.
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 23–2 for more information on the possible values for this column.
PARTITION_START	The start partition of a range of accessed partitions.
PARTITION_STOP	The stop partition of a range of accessed partitions.
PARTITION_ID	The step that has computed the pair of values of the PARTITION_START and PARTITION_STOP columns.
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 23–2 describes the values that may appear in the OTHER_TAG column.

Table 23–2 Values of OTHER_TAG Column of the PLAN_TABLE

OTHER_TAG Text	Interpretation
(blank)	Serial execution.
SERIAL_FROM_REMOTE	Serial execution at a remote site.
SERIAL_TO_PARALLEL	Serial execution; output of step is partitioned or broadcast to parallel query servers.
PARALLEL_TO_PARALLEL	Parallel execution; output of step is repartitioned to second set of parallel query servers.
PARALLEL_TO_SERIAL	Parallel execution; output of step is returned to serial “query coordinator” process.
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	Parallel execution; input of step comes from prior step in same parallel process. No interprocess communication from child.

The following table lists each combination of OPERATION and OPTION produced by the EXPLAIN PLAN command and its meaning within an execution plan.

Table 23–3 OPERATION and OPTION Values Produced by EXPLAIN PLAN

OPERATION	OPTION	Description
AND-EQUAL		An operation that accepts multiple sets of ROWIDs and returns the intersection of the sets, eliminating duplicates. This operation is used for the single-column indexes access path.
BITMAP	CONVERSION	TO ROWIDS converts the bitmap representation 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: A bitmap index full scan is performed if there is no start or stop key.
	MERGE	Merges several bitmaps resulting from a range scan into one bitmap.
	MINUS	Subtracts the bits of one bitmap from another. This row source is used for negated predicates and can be used only if there are some nonnegated predicates yielding a bitmap from which the subtraction can take place. An example appears in "Bitmap Indexes and EXPLAIN PLAN" on page 23-10.
	OR	Computes the bitwise OR of two bitmaps.
CONNECT BY		A retrieval of rows in a hierarchical order for a query containing a CONNECT BY clause.
CONCATENATION		An operation that accepts multiple sets of rows and returns the union-all of the sets.
COUNT		An operation that counts the number of rows selected from a table.
	STOPKEY	A count operation where the number of rows returned is limited by the ROWNUM expression in the WHERE clause.

Table 23–4 OPERATION and OPTION Values Produced by EXPLAIN PLAN (Continued)

OPERATION	OPTION	Description
FILTER		An operation that accepts a set of rows, eliminates some of them, and returns the rest.
FIRST ROW		A retrieval on only the first row selected by a query.
FOR UPDATE		An operation that retrieves and locks the rows selected by a query containing a FOR UPDATE clause.
HASH JOIN (These are join operations.)		An operation that joins two sets of rows, and returns the result.
	ANTI	A hash anti-join.
	SEMI	A hash semi-join.
INDEX (These operations are access methods.)	UNIQUE SCAN	A retrieval of a single ROWID from an index.
	RANGE SCAN	A retrieval of one or more ROWIDs from an index. Indexed values are scanned in ascending order.
	RANGE SCAN DESCENDING	A retrieval of one or more ROWIDs from an index. Indexed values are scanned in descending order.
INLIST ITERATOR	CONCATENATED	Iterates over the operation below it, for each value in the IN list predicate.
INTERSECTION		An operation that accepts two sets of rows and returns the intersection of the sets, eliminating duplicates.
MERGE JOIN (These are join operations.)		An operation that accepts two sets of rows, each sorted by a specific value, combines each row from one set with the matching rows from the other, and returns the result.
	OUTER	A merge join operation to perform an outer join statement.
	ANTI	A merge anti-join.
	SEMI	A merge semi-join.
CONNECT BY		A retrieval of rows in hierarchical order for a query containing a CONNECT BY clause.
MINUS		An operation that accepts two sets of rows and returns rows that appear in the first set but not in the second, eliminating duplicates.

Table 23–5 OPERATION and OPTION Values Produced by EXPLAIN PLAN (Continued)

OPERATION	OPTION	Description
NESTED LOOPS (These are join operations.)		An operation that accepts 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 and returns those rows that satisfy a condition.
	OUTER	A nested loops operation to perform an outer join statement.
PARTITION	CONCATENATED	Iterates over the operation below it, for each partition in the range given by the PARTITION_START and PARTITION_STOP columns.
PROJECTION		An internal operation.
REMOTE		A retrieval of data from a remote database.
SEQUENCE		An operation involving accessing values of a sequence.
SORT	AGGREGATE	A retrieval of a single row that is the result of applying a group function to a group of selected rows.
	UNIQUE	An operation that sorts a set of rows to eliminate duplicates.
	GROUP BY	An operation that sorts a set of rows into groups for a query with a GROUP BY clause.
	JOIN	An operation that sorts a set of rows before a merge-join.
	ORDER BY	An operation that sorts a set of rows for a query with an ORDER BY clause.
TABLE ACCESS (These operations are access methods.)	FULL	A retrieval of all rows from a table.
	CLUSTER	A 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	A retrieval of a row from a table based on its ROWID.
UNION		An operation that accepts two sets of rows and returns the union of the sets, eliminating duplicates.
VIEW		An operation that performs a view's query and then returns the resulting rows to another operation.

Note: Access methods and join operations are discussed in *Oracle8 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.

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 is as follows:

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

Formatting EXPLAIN PLAN Output

This section shows options for formatting EXPLAIN PLAN output

- How to Run EXPLAIN PLAN
- Selecting PLAN_TABLE Output in Table Format
- Selecting PLAN_TABLE Output in Nested Format

Note: The output of the EXPLAIN PLAN command reflects the behavior of the Oracle optimizer. As the optimizer evolves between releases of the Oracle server, output from the EXPLAIN PLAN command is also likely to evolve.

How to Run EXPLAIN PLAN

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 generates the following output:

```
SELECT operation, options, object_name, id, parent_id, position
  FROM plan_table
 WHERE statement_id = 'Emp_Sal'
 ORDER BY id;

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 7, 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

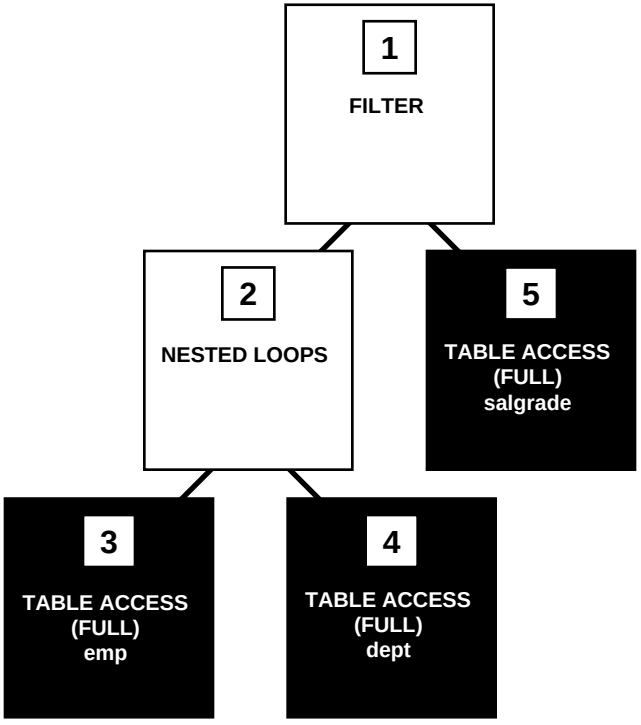
This type of SELECT statement generates a nested representation of the output that more closely depicts the processing order used for the SQL statement.

```
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, illustrated in the following figure.

Figure 23–1 Tree Structure of an Execution Plan



The tree structure illustrates how operations that occur during the execution of a SQL statement feed one another. Each step in the execution plan is assigned a number (representing the ID column of the PLAN_TABLE) and 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

EXPLAIN PLAN is not supported for statements that perform 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 type 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 24, "The SQL Trace Facility and TKPROF"

24

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
- Step 1: Set Initialization Parameters for Trace File Management
- Step 2: Enable the SQL Trace Facility
- Step 3: Format Trace Files with TKPROF
- Step 4: Interpret TKPROF Output
- Step 5: Store SQL Trace Facility Statistics
- Avoiding Pitfalls in TKPROF Interpretation
- TKPROF Output Example

Note: The SQL trace facility and TKPROF program are subject to change in future releases of the Oracle Server. Such changes will be documented as they arise.

Introduction

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
- About TKPROF
- How to Use the SQL Trace Facility and TKPROF

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

S 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.

How to Use 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 24–1 SQL Trace Facility 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 dynamic 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.
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*.

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 a Different User Session
- Enabling the SQL Trace Facility for an Instance

Attention: 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 command. For example, to issue the ALTER SESSION command 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 a Different User Session

To enable the SQL trace facility for a session other than your current session, you can call the procedure DBMS_SYSTEM.SET_SQL_TRACE_IN_SESSION. This procedure can be useful for database administrators who are not located near their users or who do not have access to the application code to set SQL trace from within an application.

This procedure requires the session ID and serial number of the user session in question, which you can obtain from the V$SESSION view. In the WHERE clause you can specify sessions by referencing the value of the OSUSER, USERNAME, or PROGRAM column in V$SESSION. For example, the following Server Manager line mode session obtains the session ID and serial number for the operating system user jausten and then enables SQL trace for that user's session:

```
SVRMGR> SELECT sid, serial#, osuser
 2>     FROM v$session
 3>     WHERE osuser = 'jausten';

SID        SERIAL#    OSUSER
---------- ---------- ---------------
         8         12 jausten
1 row selected.

SVRMGR> EXECUTE dbms_system.set_sql_trace_in_session(8,12,TRUE);
Statement processed.
```

To enable SQL trace in stored procedures, use this SQL statement:

```
DBMS_SESSION.SET_SQL_TRACE (TRUE);
```

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'
```

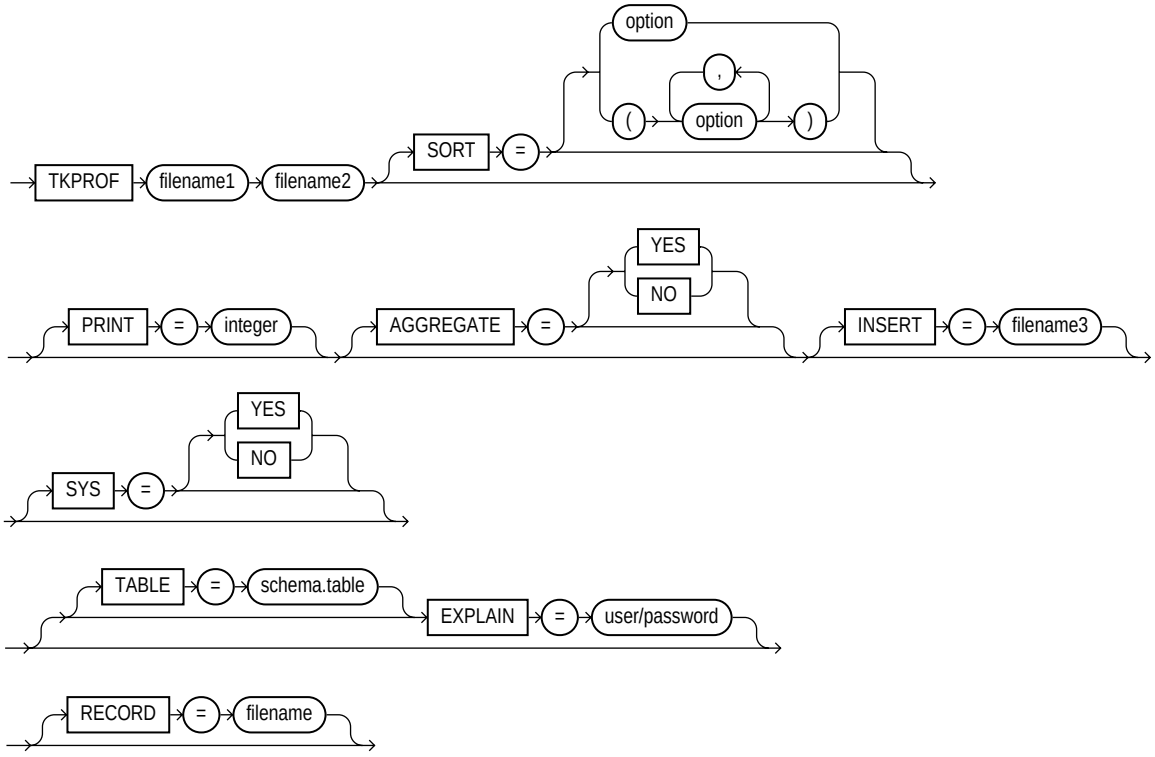
For this statement, TKPROF output includes the following information:

- 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 with no arguments, online help is displayed.

Use the following arguments with TKPROF:

Table 24–2 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.
PRINT	Lists only the first *integer* sorted SQL statements into the output file. If you omit this parameter, TKPROF lists all traced SQL statements. Note that this parameter does not affect the optional SQL script. The SQL script always inserts statistics for all traced SQL statements.
AGGREGATE	If you specify AGGREGATE = NO, then TKPROF does not aggregate multiple users of the same SQL text.
INSERT	Creates a SQL script that stores the trace file statistics in the database. TKPROF creates this script with the name *filename3*. This script creates a table and inserts a row of statistics for each traced SQL statement into the table.
SYS	Enables and disables the listing of SQL statements issued by the user SYS, or recursive SQL statements, into the output file. The default value of YES causes TKPROF to list these statements. The value of NO causes TKPROF to omit them. Note that 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 command (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 *Oracle8 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 command 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.

Table 24–2 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

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 24-26.

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 command 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. Note that 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: *Oracle8 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 command-line parameter to NO. Note that 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 24-8, 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 command line, TKPROF uses the EXPLAIN PLAN command 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 23, “The EXPLAIN PLAN Command” 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, you can see that 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. Note that 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. Note that 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. Note that 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 24-8.

```
SELECT * FROM tkprof_table;
DATE_OF_INSERT CURSOR_NUM DEPTH USER_ID PARSE_CNT PARSE_CPU PARSE_ELAP
-------------- ---------- ----- ------- --------- --------- ----------
27-OCT-1993             1     0       8         1        16         29

PARSE_DISK PARSE_QUERY PARSE_CURRENT PARSE_MISS EXE_COUNT EXE_CPU
---------- ----------- ------------- ---------- --------- -------
         3          13             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
--------- ---------- ---------- ------------ ------------- ----------
        3         26          2            2             4         14

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 23-16 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    0.11        0.21        0         0        0        0
Execute      1    0.00        0.00        0         0        0        0
Fetch        1    0.15        0.24        4       150        0        1

Misses in library cache during parse: 1
Parsing user id: 13 (DJONES)

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       0.04         0.12        0      0       0    0
Execute         1       0.01         0.01        0      0       0    0
Fetch           1       0.32         0.32       38     44       3    1

Misses in library cache during parse: 0
Parsing user id: 13  (JAUSTEN)
```

```
Rows     Execution Plan
-------  ---------------------------------------------------
      0  SELECTSTATEMENT
   3519    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      0.08       0.24        0        0       0           0
Execute        1      0.63      19.63       33      526      13           7
Fetch          0      0.00       0.00        0        0       0           0

Misses in library cache during parse: 1
Parsing user id: 13  (JAUSTEN)

Rows     Execution Plan
-------  ---------------------------------------------------
      0  UPDATE STATEMENT
   3519  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      0.01     0      0       0        0
Execute     1   0.00      0.00     0      0       0        0
Fetch       1   0.00      0.00     0      4       0        1

Misses in library cache during parse: 0
Parsing user id: 13  (JAUSTEN)

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. Note that portions have been edited out for the sake of brevity.

Header

```
Copyright (c) Oracle Corporation 1979, 1997. 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       0.15          0          0          0           0
Fetch        0      0.00       0.00          0          0          0           0
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total        1      0.00       0.15          0          0          0           0
Misses in library cache during parse: 0
Misses in library cache during execute: 1
Optimizer goal: CHOOSE
Parsing user id: 8  (SCOTT)
```

```
********************************************************************************
select emp.ename, dept.dname from emp, dept
  where emp.deptno = dept.deptno

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.12       0.14          2          0          2           0
Execute      1      0.00       0.00          0          0          0           0
Fetch        1      0.00       0.00          2          2          4          14
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total        3      0.12       0.14          4          2          6          14
Misses in library cache during parse: 1
Optimizer goal: CHOOSE
Parsing user id: 8  (SCOTT)
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         54          2          14
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total        3      0.02       0.02          1         54          2          14
Misses in library cache during parse: 1
Optimizer goal: CHOOSE
Parsing user id: 8  (SCOTT)
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         16          4           1
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total        3      0.00       0.00          0         16          4           1
Misses in library cache during parse: 1
Optimizer goal: CHOOSE
Parsing user id: 8  (SCOTT)
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          2           0
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total        3      0.00       0.00          0          1          2           0
Misses in library cache during parse: 1
Optimizer goal: CHOOSE
Parsing user id: 8  (SCOTT)
```

```
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          2          4          15
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total        3      0.00       0.00          0          2          4          15
Misses in library cache during parse: 1
Optimizer goal: CHOOSE
Parsing user id: 8  (SCOTT)
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      0.06       0.08          2         18          1           0
Execute      1      0.00       0.00          0          0          0           0
Fetch        1      0.01       0.01          1         11         12          14
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total        3      0.07       0.09          3         29         13          14
Misses in library cache during parse: 1
Optimizer goal: CHOOSE
```

```
Parsing user id: 8  (SCOTT)
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: 8  (SCOTT)
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: 8  (SCOTT)
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: 8  (SCOTT)
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       0.04          0          2          3          12
Fetch        0      0.00       0.00          0          0          0           0
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total        2      0.02       0.04          0          2          3          12
Misses in library cache during parse: 1
Optimizer goal: CHOOSE
Parsing user id: 8  (SCOTT)
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          12
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total        3      0.01       0.01          0          1          2          12
Misses in library cache during parse: 1
Optimizer goal: CHOOSE
Parsing user id: 8  (SCOTT)
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.
```

25

Using Oracle Trace

This chapter describes how to use Oracle Trace to collect Oracle Server event data. It covers:

- Introduction
- Using Oracle Trace for Server Performance Data Collection
- Using Initialization Parameters to Control Oracle Trace
- Using Stored Procedure Packages to Control Oracle Trace
- Using the Oracle Trace Command-Line Interface
- Oracle Trace Collection Results

Introduction

Oracle Trace is a general-purpose data collection product that has been introduced with the Oracle Enterprise Manager systems management product family. You can use the Oracle Trace data collection API in any software product to collect data for a variety of uses, such as performance monitoring, diagnostics, and auditing. Oracle Trace collects specific data for events defined within the host product.

The server performance data that you can collect with Oracle Trace includes:

- SQL statements and statistics on the frequency, duration, and resources used for all parse, execution, and fetch events
- Execution plan details
- Logical and physical database transactions, including the frequency, duration, and resources used by each transaction event
- A set of statistics associated with each of these duration events, including: UGA and PGA memory, block changes, block gets, consistent gets, physical reads, redo entries, redo size, sorts in memory and disk
- Resource usage for each database event measured in CPU time, file I/Os and page faults

See Also: *Oracle Trace User's Guide* and *Oracle Trace Developer's Guide* contained in the Oracle Enterprise Manager Performance Pack documentation set. These books contain a complete list of events and data that can be collected for Oracle Server.

Using Oracle Trace for Server Performance Data Collection

You can use Oracle Trace to collect server performance data for a specific database session or for the entire instance. You can also select the server *event set* for which you want to collect.

Oracle Trace lets you organize host application events into event sets. Doing so allows you to collect data for a specific subset of all potential host application events. Oracle has defined the following event sets: ALL, DEFAULT, and EXPERT. The ALL set includes all server events, the DEFAULT set excludes server WAIT events, and the EXPERT set is specifically defined for use in the Oracle Expert tuning application. Oracle recommends using the DEFAULT event set.

You can enable and control server collections in the following ways:

- Using the Oracle Trace Manager application, a Windows-based graphical user interface that is supplied with the Oracle Enterprise Manager Performance Pack (a database option you can license)
- Using nongraphical, server-based controls:
  - Oracle Trace database initialization parameters
  - Oracle Trace stored procedure packages
  - Oracle Trace Command-Line Interface

The following sections describe the server-based controls.

See Also: *Oracle Trace User's Guide*

Using Initialization Parameters to Control Oracle Trace

Six parameters are set up by default to control Oracle Trace. By logging into the internal account in your database and executing a SHOW PARAMETERS TRACE command, you will see the following parameters::

Table 25–1 Oracle Trace Initialization Parameters

Name	Type	Value
ORACLE_TRACE_COLLECTION_NAME	string	[null]
ORACLE_TRACE_COLLECTION_PATH	string	$ORACLE_HOME/rdbms/log
ORACLE_TRACE_COLLECTION_SIZE	integer	5242880
ORACLE_TRACE_ENABLE	boolean	FALSE
ORACLE_TRACE_FACILITY_NAME	string	oracled
ORACLE_TRACE_FACILITY_PATH	string	ORACLE_HOME/rdbms/admin

You can modify the Oracle Trace and use them by adding them to your initialization file.

Note: This chapter references file pathnames 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 *Oracle8 Reference*.

Enabling Oracle Trace Collections

Note that 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, the parameter should be set to TRUE. Setting the parameter to TRUE does not start an Oracle Trace collection, but allows Oracle Trace to be used for that server. Oracle Trace can then be started in one of the following ways:

- Using the Oracle Trace Manager application (supplied with the OEM Performance Pack)
- Setting the ORACLE_TRACE_COLLECTION_NAME parameter

When ORACLE_TRACE_ENABLE is set to TRUE, you can initiate an Oracle Trace server collection by entering 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 long. You must then shut down your database and start it up again before the parameters take effect. If a collection name is specified, when you start the server, you automatically start an Oracle Trace collection for all database sessions.

To stop the collection, 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 Which Oracle Trace Collects

The ORACLE_TRACE_FACILITY_NAME determines the event set that Oracle Trace will collect. The name of the DEFAULT event set is ORACLED. The ALL event set is ORACLE and the EXPERT event set is ORACLEE.

If, once restarted, the database does not start collecting data, you should check the following:

- The event set file, identified by the ORACLE_TRACE_FACILITY_NAME (with .fdf appended to it) should be located in the directory identified by ORACLE_TRACE_FACILITY_PATH.
- 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 Oracle Trace parameters should be set to the values that you changed in the initialization file.
- Look for an EPC_ERROR.LOG file. It will give you more information about why a collection may have failed.

Using Stored Procedure Packages to Control Oracle Trace

Using the Oracle Trace stored procedure packages 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:

```
DBMS_ORACLE_TRACE_USER.SET_ORACLE_TRACE(TRUE/FALSE, collection_name,
server_event_set)
```

- true / false = Boolean: TRUE to turn on, FALSE to turn off
- collection_name = varchar2: collection name (no file extension, 8 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:

```
DBMS_ORACLE_TRACE_AGENT.SET_ORACLE_TRACE_IN_SESSION(sid, serial#, true/false,
collection_name, server_event_set)
```

- sid = number: session instance from v$session.sid
- serial# = number: session serial number from v$session.serial#

Example:

```
EXECUTE DBMS_ORACLE_TRACE_USER.SET_ORACLE_TRACE_IN_SESSION (8,12,TRUE,"MYCOLL",
"oracled");
```

If the collection does not occur, you should 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, you must run the OTRCSVR.SQL file (in your Oracle Trace admin directory) to create the packages.

Using the Oracle Trace Command-Line Interface

Another option for controlling Oracle Trace server collections is the Oracle Trace command-line interface (CLI). 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 coll_name cdf_file
- OTRCCOL DFD coll_name username password service

The parameter JOB_ID can be any numeric value. (You must remember this value in order to stop the collection.) The input parameter file contains specific parameter values required for each function. Examples follow. COLL_NAME (collection name) and CDF_FLE (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 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, ORACLED, and ORACLEE. See "Using Initialization Parameters to Control Oracle Trace" on page 25-4 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 as follows:

```
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 "Formatting Oracle Trace Data to Oracle Tables" on page 25-10 for information on formatting part or all of an Oracle Trace collection, and 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.

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 two ways:

- You can create Oracle Trace Detail Reports from the binary file.
- The data can be formatted to Oracle tables for SQL access and reporting.

Oracle Trace Detail Reports

Oracle Trace Detail Reports display 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 Detail Report:

```
OTRCREP  [optional parameters]  collection_name.CDF
```

The first step that you may want to take is to run a report called PROCESS.txt. You can produce this report first to give you a listing of specific process identifiers for which you want to run the detail report.

The command parameter used to produce a Process report is:

```
OTRCREP  -P  collection_name.CDF
```

Other optional detail report parameters are:

output_path	specifies a full output path for the report files. If not specified, the files will be placed in the current directory
-p	creates a report for a specific process ID obtained from the PROCESS report. For example, a detail report for process 1234 would use -p1234
-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
-a	creates a report containing all the events for all products, in the order they occur in the data collection (.dat) file. The report is a text display of all items for all events.

Formatting Oracle Trace Data to Oracle Tables

Your Oracle Trace server collection can be formatted to Oracle tables for more flexible access by any SQL reporting tool. Oracle Trace will produce a separate table for each event collected. For example, a “parses” event table is created to store data for all parse events that occur 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.

Use the following command to format an Oracle Trace collection:

```
OTRCFMT [optional parameters] collection_name.cdf [user/password@database]
```

If user/password@database is omitted, the user will be prompted 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*

26

Registering Applications

Application developers can use the DBMS_APPLICATION_INFO package with Oracle Trace and the SQL trace facility to record the name of the executing module or transaction in the database for use later 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. Topics in this chapter include:

- Overview
- Registering Applications
- Setting the Module Name
- Setting the Action Name
- Setting the Client Information
- Retrieving Application Information

Overview

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.

Registering Applications

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

DBMS_APPLICATION_INFO Package

DBMS_APPLICATION_INFO provides the following procedures:

Table 26–1 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. For more information about Oracle supplied packages and executing stored procedures, see the *Oracle8 Application Developer's Guide*.

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 following sample PL/SQL block 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.

Index

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W