Oracle9*i*

Database Performance Methods

Release 1 (9.0.1)

June 2001

Part No. A87504-02

ORACLE®

Oracle9i Database Performance Methods, Release 1 (9.0.1)

Part No. A87504-02

Primary Author: Michele Cyran

Contributing Author: Andrew Holdsworth

Contributors: Jorn Bartels, Maria Colgan, Bjorn Engsig, Cecilia Gervasio, Connie Dialeris Green, Mattias Jankowitz, Peter Kilpatrick, Anjo Kolk, JP Polk, Virag Saksena, Sabrina Whitehouse, Graham Wood

Graphic Designer: Valarie Moore

Contents

Send Us Your Comments

Oracle9i Database Performance Methods, Release 1 (9.0.1)

Part No. A87504-02

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

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

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

- Electronic mail: infodev_us@oracle.com
- FAX: (650) 506-7227   Attn: Server Technologies Documentation Manager
- Postal service:
  Oracle Corporation
  Server Technologies Documentation
  500 Oracle Parkway, Mailstop 4op11
  Redwood Shores, CA 94065
  USA

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

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

Preface

This book describes ways to improve Oracle performance by starting with good application design and using statistics to monitor application performance. It explains the Oracle Performance Improvement Method and the Emergency Performance Method, and it provides techniques for dealing with performance problems.

This preface contains these topics:

Audience

Oracle9i Database Performance Methods is a high-level aid for people responsible for the operation, maintenance, and performance of Oracle. To use this book, you could be a database administrator, application designer, programmer, or manager. You should be familiar with Oracle9*i*, the operating system, and application design before reading this manual.

Organization

This document contains:

Chapter 1, "Designing and Developing for Performance"

This chapter describes performance issues to consider when designing Oracle applications.

Chapter 2, "Monitoring and Improving Application Performance"

This chapter describes the Oracle Performance Improvement Method and the importance of statistics for application performance improvements.

Chapter 3, "Emergency Performance Techniques"

This chapter describes techniques for dealing with performance emergencies.

Related Documentation

Before reading this manual, you should have already read *Oracle9i Database Concepts*, the *Oracle9i Application Developer's Guide - Fundamentals*, and the *Oracle9i Database Administrator's Guide.*

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

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

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

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

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

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

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

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

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

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

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

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

Conventions

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

- Conventions in Text
- Conventions in Code Examples

Conventions in Text

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

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

Conventions in Code Examples

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

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

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

Convention	Meaning	Example
[ ]	Brackets enclose one or more optional items. Do not enter the brackets.	DECIMAL (digits [, precision])
{ }	Braces enclose two or more items, one of which is required. Do not enter the braces.	{ENABLE | DISABLE}
|	A vertical bar represents a choice of two or more options within brackets or braces. Enter one of the options. Do not enter the vertical bar.	{ENABLE | DISABLE} [COMPRESS | NOCOMPRESS]
...	Horizontal ellipsis points indicate either:	
	■ That we have omitted parts of the code that are not directly related to the example	CREATE TABLE ... AS subquery;
	■ That you can repeat a portion of the code	SELECT col1, col2, ... , col*n* FROM employees;
. . .	Vertical ellipsis points indicate that we have omitted several lines of code not directly related to the example.	
Other notation	You must enter symbols other than brackets, braces, vertical bars, and ellipsis points as it is shown.	acctbal NUMBER(11,2); acct CONSTANT NUMBER(4) := 3;
Italics	Italicized text indicates variables for which you must supply particular values.	CONNECT SYSTEM/*system_password*
UPPERCASE	Uppercase typeface indicates elements supplied by the system. We show these terms in uppercase in order to distinguish them from terms you define. Unless terms appear in brackets, enter them in the order and with the spelling shown. However, because these terms are not case sensitive, you can enter them in lowercase.	SELECT last_name, employee_id FROM employees; SELECT * FROM USER_TABLES; DROP TABLE hr.employees;
lowercase	Lowercase typeface indicates programmatic elements that you supply. For example, lowercase indicates names of tables, columns, or files.	SELECT last_name, employee_id FROM employees; sqlplus hr/hr

Documentation Accessibility

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

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

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

1 Designing and Developing for Performance

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

This chapter contains the following subjects:

Oracle's New Methodology

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

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

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

See Also: *Oracle9i Database Performance Guide and Reference*

Understanding Investment Options

With the availability of relatively inexpensive, high-powered processors, memory, and disk drives, there is a temptation to buy more system resources to improve performance. In many situations, new CPUs, memory, or more disk drives can indeed provide an immediate performance improvement. However, any performance increases achieved by adding hardware should be considered a short-term relief to an immediate problem. If the demand and load rates on the application continue to grow, then the chance that you will face the same problem in the near future is very likely.

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

Understanding Scalability

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

What is Scalability?

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

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

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

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

Examples of resource exhaustion include the following:

- Hardware exhaustion
- Table scans in high-volume transactions causing inevitable disk I/O shortages
- Excessive network requests, resulting in network and scheduling bottlenecks
- Memory allocation causing paging and swapping
- Excessive process and thread allocation causing operating system thrashing

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

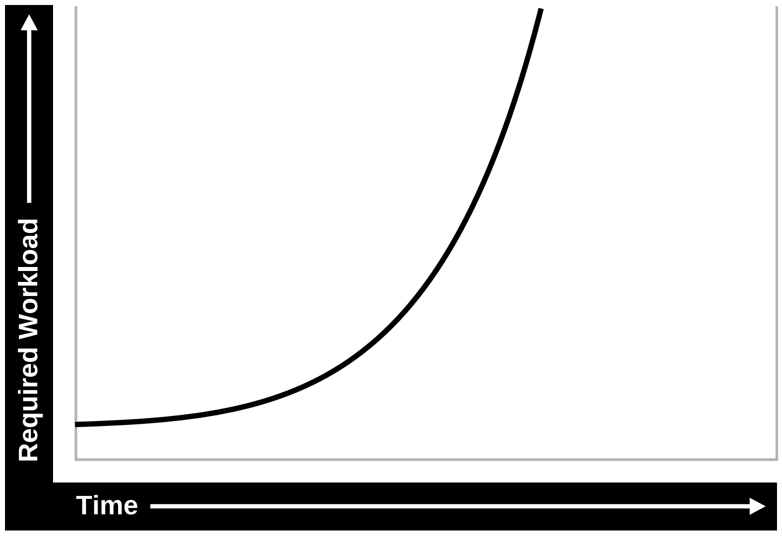
Internet Scalability

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

Characteristics of Internet age applications include the following:

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

Figure 1–1 Internet Growth Curve



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

Internet applications are challenged by very short development timeframes with limited time for testing and evaluation. However, bad design generally means that at some point in the future, the system will need to be re-architected or re-implemented. If an application with known architectural and implementation limitations is deployed on the Internet, and if the workload exceeds the anticipated demand, then there is real chance of failure in the future. From a business perspective, poor performance can mean a loss of customers. If Web users do not get a response in seven seconds, then the user’s attention could be lost forever.

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

Factors Preventing Scalability

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

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

Factors Preventing Linear Scalability

1. Poor Application Design, Implementation, and Configuration

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

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

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

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

2. Incorrect Sizing of Hardware Components

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

3. Limitations of Software Components

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

4. Limitations of Hardware Components

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

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

System Architecture

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

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

Hardware and Software Components

Hardware Components

Today's designers and architects are responsible for sizing and capacity planning of hardware at each tier in a multi-tier environment. It is the architect's responsibility to achieve a balanced design. This is analogous to a bridge designer who

must consider all the various payload and structural requirements for the bridge. A bridge is only as strong as its weakest component. As a result, a bridge is designed in balance, such that all components reach their design limits simultaneously.

The main hardware components are the following:

- CPU
- Memory
- I/O Subsystem
- Network

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

Memory Database and application servers require considerable amounts of memory to cache data and avoid time-consuming disk access.

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

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

> **See Also:** *Oracle9i Database Performance Guide and Reference* for more information on tuning these resources

Software Components

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

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

Most applications involve the following components:

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

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

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

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

Common functions of this component include the following:

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

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

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

Common functions of this component include the following:

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

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

Common functions of this component include the following:

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

Configuring the Right System Architecture for Your Requirements

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

Examples of interactive user applications:

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

Examples of process-driven applications:

- Utility billing systems
- Fraud detection systems
- Direct mail

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

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

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

- How many users will the system support?

  Most applications fall into one of the following categories:

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

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

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

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

  – An infinite user population distributed on the Internet

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

- What will be the user interaction method?

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

- Where are the users located?

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

- What is the network speed?

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

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

  The amount of data queried online influences all aspects of the design, from table and index design to the presentation layers. Design efforts must ensure that user response time is not a function of the size of the database. If the application is largely read only, then replication and data distribution to local caches in the application servers become a viable option. This also reduces workload on the core transactional server.

- What is the user response time requirement?

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

- Do users expect 24 hour service?

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

- Must all changes be made in real time?

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

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

- How big will the database be?

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

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

Application Design Principles

This section describes design decisions that are involved in building applications.

Simplicity In Application Design

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

Consider some of the following design issues:

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

Data Modeling

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

Table and Index Design

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

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

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

Index design is also a largely iterative process, based on the SQL generated by application designers. However, it is possible to make a sensible start by building indexes that enforce primary key constraints and indexes on known access patterns, such as a person's name. As the application evolves and testing is performed on realistic sizes of data, certain queries will need performance improvements for which building a better index is a good solution. The following list of indexing design ideas should be considered when building a new index:

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

Appending Columns to an Index or Using Index-Organized Tables

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

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

Using a Different Index Type

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

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

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

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

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

See Also: *Oracle9i Database Performance Guide and Reference*

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

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

Finding the Cost of an Index

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

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

Serializing within Indexes

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

Ordering Columns in an Index

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

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

Using Views

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

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

SQL Execution Efficiency

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

- Good Database Connection Management

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

- Good Cursor Usage and Management

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

  - **Hard Parsing**. A SQL statement is submitted for the first time, and no match is found in the shared pool. Hard parses are the most resource-intensive and unscalable, because they perform all the operations involved in a parse.

  - **Soft Parsing**. A SQL statement is submitted for the first time, and a match *is* found in the shared pool. The match can be the result of previous execution by another user. The SQL statement is shared, which is good for performance. However, soft parses are not ideal, because they still require syntax and security checking, which consume system resources.

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

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

Statement with string literals:

```
SELECT * FROM emp
WHERE ename
LIKE 'KING';
```

Statement with bind variables:

```
SELECT * FROM emp
WHERE ename
LIKE :1;
```

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

```
Test                            #Users Supported

No Parsing all statements              270

Soft Parsing all statements            150

Hard Parsing all statements             60

Re-Connecting for each Transaction      30
```

These tests were performed on a four-CPU machine. The differences increase as the number of CPUs on the system increase.

Implementing the Application

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

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

- User Interface

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

- Business Logic

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

- User Requests and Resource Allocation

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

- Data Management and Transactions

  Most of this is not affected by the programming language.

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

code. There is great temptation to put procedural logic into the SQL access. This tends to result in poor SQL that is resource-intensive. SQL statements with `DECODE` case statements are very often candidates for optimization, as are statements with a large amount of `OR` predicates or set operators, such as `UNION`, `MINUS`, etc.

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

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

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

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

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

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

Trends in Application Development

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

Java provides better portability of code and availability to programmers. However, there are a number of performance implications associated with Java. Because Java is an interpreted language, it is slower at executing similar logic than compiled languages such as C. As a result, resource usage of client machines increases. This requires more powerful CPUs to be applied in the client or middle-tier machines and greater care from programmers to produce efficient code.

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

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

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

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

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

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

Workload Testing, Modeling, and Implementation

Sizing Data

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

When the system becomes operational it becomes harder to predict database growth, especially that of indexes. Tables grow over time, and indexes are subject to the individual behavior of the application in terms of key generation, insertion pattern, and deletion of rows. The worst case is where you insert using an ascending key and then delete most rows from the left-hand side but not all the rows. This leaves gaps and wasted space. If you have index use like this make sure that you know how to use the online index rebuild facility.

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

Estimating Workloads

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

Extrapolating From a Similar System

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

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

Benchmarking

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

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

Benchmarks by their nature stress all system components to their limits. As all components are being stressed be prepared to see all errors in application design and implementation manifest themselves while benchmarking. Benchmarks also test database, operating system, and hardware components. Because most benchmarks are performed in a rush, expect setbacks and problems when a system component fails. Benchmarking is a stressful activity, and it takes considerable experience to get the most out of a benchmarking exercise.

Application Modeling

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

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

Testing, Debugging, and Validating a Design

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

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

- Test with realistic data volumes and distributions.

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

- Use the correct optimizer mode.

  All testing should be performed with the optimizer mode that will be used in production. All Oracle research and development effort is focused upon the cost-based optimizer, and for this reason we recommend the use of the cost-based optimizer.

- Test a single user performance.

  A single user on an idle or lightly used system should be tested for acceptable performance. If a single user cannot get acceptable performance under ideal conditions, it is impossible there will be good performance under multiple users where resources are shared.

- Obtain and document plans for all SQL statements.

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

- Attempt multi-user testing.

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

- Test with the correct hardware configuration.

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

- Measure steady state performance.

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

Deploying New Applications

This section describes design decisions involved deploying applications.

Rollout Strategies

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

- Big Bang Approach - All users migrate to the new system at once.
- Trickle Approach - Users slowly migrate from existing systems to the new one.

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

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

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

Performance Checklist

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

1. When you create the production database, allow for growth by setting `MAXINSTANCES`, `MAXDATAFILES`, `MAXLOGFILES`, `MAXLOGMEMBERS`, and `MAXLOGHISTORY` to values above what is anticipated for the rollout. This results in more disk space usage and bigger control files, but saves time later should these need extension in an emergency.
2. Set block size and optimizer mode to that used to develop the application. Export the schema statistics from the development/test environment to the production database if the testing was done on representative data volumes and the current SQL execution plans are correct.
3. Set the minimal number of initialization parameters. The important parameters to set size the various caches within the SGA. The additional parameters that specify the behavior of the archive dump destinations should be set for backup and debugging purposes. Ideally, most other parameters should be left at default. If there is more tuning to perform, this shows up when the system is under load.

   See Also: *Oracle9i Database Performance Guide and Reference* for guidance on setting minimal parameters in initial instance configuration

4. Be prepared to manage block contention by setting storage options of database objects. Tables and indexes that experience high `INSERT`/`UPDATE`/`DELETE` rates should be created with either automatic segment space management or multiple freelists and an increased setting of `INITRANS`. To avoid contention of rollback segments, either automatic undo management should be used or

multiple rollback segments should be created to support the required user population.

> **See Also:** *Oracle9i Database Administrator's Guide* for more information on using automatic undo management and on managing free space with automatic segment space management

5. All SQL statements should be verified to be optimal and their resource usage understood.
6. Validate that middleware and programs that connect to the database are efficient in their connection management and do not logon/logoff repeatedly.
7. Validate that the SQL statements use cursors efficiently. Each SQL statement should be parsed once and then executed multiple times. The most common reason this does not happen is because bind variables are not used properly and `WHERE` clause predicates are sent as string literals. If the precompilers are used to develop the application, then make sure that the parameters `MAXOPENCURSORS`, `HOLD_CURSOR`, and `RELEASE_CURSOR` have been reset from the default values prior to precompiling the application.
8. Validate that all schema objects have been correctly migrated from the development environment to the production database. This includes tables, indexes, sequences, triggers, packages, procedures, functions, java objects, synonyms, grants, and views. Ensure that any modifications made in testing are made to the production system.
9. As soon as the system is rolled out, establish a baseline set of statistics from the database and operating system. To do this, use Enterprise Manager or Statspack. This first set of statistics validates or corrects any assumptions made in the design and rollout process.
10. Start anticipating the first bottleneck (there will always be one) and follow the Oracle performance method to make performance improvement.

2 Monitoring and Improving Application Performance

This chapter contains the following subjects:

- Importance of Statistics
- The Oracle Performance Improvement Method

Importance of Statistics

Before reacting to a problem, collect all possible statistics and get an overall picture of the application. Getting a complete landscape of the system may take considerable effort. But, if data has already been collected and embedded into the application,thenthisprocessismuch easier.

After collecting as much initial data as possible, outline issues found from the statistics, the same way doctors collect symptoms from patients. Reacting to symptoms too early in the performance analysis process generally results in an incorrect analysis, which wastes time later. For example, it is extremely risky for a doctor to prescribe open heart surgery for a patient who complains of chest pains on the initial consultation.

Operating System Statistics

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

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

Operating system statistics include the following:

- CPU Statistics
- Virtual Memory Statistics
- Disk Statistics
- Network Statistics

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

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

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

Note: On UNIX systems, where wait for I/O is derived for part of the CPU statistics, this value should be treated as idle time.

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

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

Disk Statistics Because the database resides on a set of disks, the performance of the I/O subsystem is very important to the performance of the database. Most operating systems provide extensive statistics on disk performance. The most important disk statistics are the current response time and the length of the disk queues. These statistics show if the disk is performing optimally or if the disk is being overworked. If a disk shows response times over 20 milliseconds, then it is performing badly or is overworked. This is your bottleneck. If disk queues start to exceed two, then the disk is a potential bottleneck of the system.

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

Database Statistics

Database statistics provide information on the type of load on the database, as well as the internal and external resources used by the database. When database resources become exhausted, it is possible to identify bottlenecks in the application.

Database statistics can be queried directly from the database in a relational manner using SQL. These statistics can be inserted back into the database with the `INSERT INTO` x AS SELECT ... or CREATE TABLE x AS SELECT ... statements. This is the basis of most snapshot mechanisms that allow statistical gathering over time. Most statistics are contained in a series of virtual tables or views known as the V$ tables, because they are prefixed with V$. These are read only, and they are owned by SYS. Many of the tables contain identifiers and keys that can be joined to other V$ tables.

In order to get meaningful database statistics, the `TIMED_STATISTICS` parameter must be enabled for the database instance. The performance impact of having `TIMED_STATISTICS` enabled is minimal compared to instance performance. The performance improvements and debugging value of a complete set of statistics make this parameter crucial to effective performance analysis.

The core database statistics are:

- Buffer Cache
- Shared Pool
- Wait Events

Buffer Cache The buffer cache manages blocks read from disk into buffers in memory. It also holds information on the most recently used buffers and those modified in normal database operation. To get best query performance, a user query accesses all required data blocks within the buffer cache, thus satisfying the query from memory. However, this might not always happen, because the database is many multiples the size of the buffer cache. With this in mind, it is easy to see that the buffer cache requires management and tuning.

The objective in tuning the buffer cache is to get acceptable user query time by having as many of the required blocks in the cache as possible. Also, eliminate time consuming I/Os without inducing any serialization points or performance spikes as old blocks are aged out of the cache. This process requires a working knowledge of the buffer cache mechanism, the database writer, and the checkpointing mechanism. Most information can be extracted from the V$SYSSTAT table.

Shared Pool The shared pool contains information about user sessions, shared data structures used by all database sessions, and the dictionary cache.

Querying the shared pool allows analysis of the SQL statements run in the database. This is particularly important if you have limited or no knowledge of the application source code. In addition to the actual SQL, you can determine how many times it is run and how much CPU and disk I/Os are performed by the SQL. This information can be extracted from the `V$SQL` table. Analyzing this information is crucial in objective bottleneck identification when debugging an unknown application.

Wait Events In the process of usual database server operations, there are times when processes need to share resources and/or synchronize with other processes; for example, allocating memory in the shared pool or waiting for a lock. Similarly, there are times when the database process gives control to external code or other processes out of its control; for example, performing I/O and waiting for the log writer to synchronize the redo log.

In these cases, the user process stops working and starts waiting. This wait time becomes part of the eventual user response time. If there are multiple processes queuing on a shared resource or demanding the same external resource, then the database starts to single-thread, and scalability is impacted. Performance analysis should determine why queuing on resources in the database is happening.

The `V$SYSTEM_EVENT`, `V$SESSION_EVENT`, and `V$SESSION_WAIT` tables allow querying of historical wait events or wait events in real time. The `V$SESSION_WAIT` table has additional columns that can be joined to other `V$` tables based on the wait event recorded. These additional join columns specified in `V$SESSION_WAIT` allow focused drill down and analysis of the wait event.

> **See Also:** *Oracle9i Database Reference* for reference information on the V$ tables

Application Statistics

Application statistics are probably the most difficult statistics to get, but they are the most important statistics in measuring any performance improvements made to the system. At a minimum, application statistics should provide a daily summary of user transactions processed per working period. More complete statistics provide precise details of what transactions were processed and the response times for each transaction type. Detailed statistics also provide statistics on the decomposition of each transaction time spent in the application server, the network, the database, etc.

The best statistics require considerable instrumentation of the application. This is best built into the application from the start, because it is difficult to retrofit into existing applications.

Statistics Gathering Tools

Operating System Data Gathering Tools

Table 2–1 shows the various tools for gathering operating statistics on UNIX.

Table 2–1 UNIX Tools for Operating Statistics

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

For Windows NT/2000, use the Performance Monitor tool.

Database Data Gathering Tools

Oracle provides three primary data gathering tools. These tools are increasingly more complex to install and run. However, as they increase in complexity, they provide better reporting output. The tools are:

1. BSTAT/ESTAT scripts

   These scripts are located in the `$ORACLE_HOME/rdbms/admin` directory in files UTLBSTAT.SQL and UTLESTAT.SQL. They produce a simple report of database activity between two points in time. The reports can then be archived over time. These statistics represent the bare minimum of statistics that should be kept.

2. Statspack

   Statspack builds on the BSTAT/ESTAT scripts, but it extends the data capture to store all statistics in a database repository, which allows better baseline setting and offline analysis. The statspack report provides considerably more information than BSTAT/ESTAT in a format useful for bottleneck detection. This mechanism is the best way to record and collect database statistics.

3. Oracle Enterprise Manager (EM)

   Oracle Enterprise Manager provides a graphical user interface for collecting, storing, and reporting performance data. The EM Intelligent Agent data gathering service can collect this performance data on a scheduled basis. A single agent can manage the data collections for all Oracle databases and the operating system of the target node. The data is automatically stored in an historical data repository for performance reporting. Data stored in the repository can be used to analyze many facets of database performance, such as database load, cache allocations and efficiency, resource contention, and high-impact SQL.

Performance data collections can be initiated directly from the EM Console or through the EM Diagnostics Pack - Capacity Planner application. HTML reports of historical performance data can be generated from the EM Console. These reports provide a comprehensive analysis of database system usage and performance, which can be easily accessed and navigated from a browser. EM also provides a graphical real-time Performance Overview for monitoring a subset of these performance metrics using line charts, bar graphs, and so forth.

The Performance Overview charts let you troubleshoot existing performance problems by drilling into performance data to track down the source of a performance bottleneck. For example, a decline in the memory sort percentage can be immediately investigated by drilling down to the sessions and corresponding SQL responsible for high-volume sort activity. High-impact SQL statements discovered through this process can be further investigated by launching SQL diagnostic tools in the context of the problem.

See Also: *Oracle Enterprise Manager Concepts Guide*

Importance of Historical Data and Baselines

One of the biggest challenges for performance engineers is determining what changed in the system to cause a satisfactory application to start having performance problems. The list of possibilities on a modern complex system is extensive.

Historical performance data is crucial in eliminating as many variables as possible. This means that you should collect operating system, database, and application statistics from the first day an application is rolled out into production. This applies even if the performance is unsatisfactory. As the applications stabilize and the performance characteristics are better understood, a set of statistics becomes the baseline for future reference. These statistics can be used to correlate against a day when performance is not satisfactory. They are also essential for future capacity and growth planning.

Performance Intuition

Database and operating system statistics provide an indication of how well a system is performing. By correlating statistics with actual throughput, you can see how the system is performing and determine where future bottlenecks and resource shortages could exist. This is a skill acquired through the experience of monitoring systems and working with the Oracle server.

CPU utilization is the easiest system usage statistic to understand and monitor. Monitoring this statistic over time, you can see how the system is used during the work day and over a number of weeks. However, this statistic provides no indication of how many business transactions were executed or what resources were used per transaction.

Two other statistics that give a better indication of actual business transactions executed are the number of commits and the volume of redo generated. These are found in the `V$SYSSTAT` view under `USER COMMITS` and `REDO SIZE`. These statistics show the number of actual transactions and the volume of data changed in the database. If these statistics increase in number over time, and if application and transaction logic are not altered, then you know that more business transactions were executed. The number of logical blocks read (`V$SYSSTAT` statistic `'session logical reads'`) also indicates the query workload on a system. Be careful interpreting this number. A change in the number of logical blocks read can be a result of an execution plan change rather than an increase in workload.

With experience, it becomes easy to correlate database statistics with the application workload. A performance DBA learns to use intuition with database statistics and the application profile to determine a system's workload characteristics. A DBA must also anticipate the expected performance of frequently executed transactions. Understanding the core SQL statements in an application is key to performance diagnosis. Much of this activity can be done informally.

For example, a core business transaction is required to run in a subsecond response time. Initial investigation of the transaction shows that this transaction performs 200 logical reads, of which 40 are always obtained from disk. Taking a disk response time of 20 milliseconds, the likely I/O time is 40 x .02 = 0.8 seconds, which probably fails the response time target. The DBA requests that the transaction be rewritten, and the number of logical I/Os is reduced to 80, with an average of five coming from disk.

To avoid poor performance, it is wise to perform this type of calculation before production roll out. The process should be repeated after the system is in production, because the data volumes grow and the transaction statistics can change.

The Oracle Performance Improvement Method

Introduction to Performance Improvement

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

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

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

Before looking at any database or operating system statistics, it is crucial to get feedback from the most important components of the system: the users of the system and the people ultimately paying for the application. Examples of typical user feedback are as follows:

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

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

Examples of real goals for the examples quoted above could be as follows:

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

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

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

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

Steps in The Oracle Performance Improvement Method

1. Get candid feedback from users. Determine the performance project’s scope and subsequent performance goals, as well as performance goals for the future. This process is key in future capacity planning.
2. Get a full set of operating system, database, and application statistics from the system when the performance is both good and bad. If these are not available, then get whatever is available. Missing statistics are analogous to missing evidence at a crime scene: They make detectives work harder and it is more time-consuming.

3. Sanity-check the operating systems of all machines involved with user performance. By sanity-checking the operating system, you look for hardware or operating system resources that are fully utilized. List any over-used resources as symptoms for analysis later. In addition, check that all hardware shows no errors or diagnostics.

4. Check for the top ten most common mistakes with Oracle, and determine if any of these are likely to be the problem. List these as symptoms for analysis later. These are included because they represent the most likely problems.

   See Also: "Top Ten Mistakes Made by Oracle Users" on page 2-13

5. Build a conceptual model of what is happening on the system using the symptoms as clues to understand what caused the performance problems.

   See Also: "A Sample Decision Process for Performance Conceptual Modeling" on page 2-12

6. Propose a series of remedy actions and the anticipated behavior to the system, and apply them in the order that can benefit the application the most. A golden rule in performance work is that you only change one thing at a time and then measure the differences. Unfortunately, system downtime requirements might prohibit such a rigorous investigation method. If multiple changes are applied at the same time, then try to ensure that they are isolated.

7. Validate that the changes made have had the desired effect, and see if the user's perception of performance has improved. Otherwise, look for more bottlenecks, and continue refining the conceptual model until your understanding of the application becomes more accurate.

8. Repeat the last three steps until performance goals are met or become impossible due to other constraints.

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

How to Check the Operating System

This section defines the process for collecting operating system symptoms.

1. Check CPU utilization in user and kernel space for the total system and on each CPU.
2. Confirm that there is no paging or swapping.
3. Check that network latencies between machines are acceptable.
4. Find disks with poor response times or long queues.
5. Confirm that there are no hardware errors.

A Sample Decision Process for Performance Conceptual Modeling

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

This section illustrates how a performance engineer might look for bottlenecks. Use this only as a guideline for the process. With experience, performance engineers add to the steps involved. This analysis assumes that statistics for both the operating system and the database have been gathered.

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

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

2. Is all the CPU being utilized?

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

Server releases prior to 9*i*, use buffer gets as the measure for CPU usage. With release 9*i*, Oracle provides the actual CPU statistics in `V$SQL`.

> **See Also:** *Oracle9i Database Reference* for more information on `V$SQL`

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

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

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

Top Ten Mistakes Made by Oracle Users

This section lists the most common mistakes found in Oracle systems. By following Oracle's performance improvement methodology, you should be able to avoid these mistakes altogether. If you find these mistakes in your system, then re-engineer the application where the performance effort is worthwhile.

> **See Also:** *Oracle9i Database Performance Guide and Reference* for more information on wait events

1. Bad Connection Management

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

2. Bad Use of Cursors and the Shared Pool

   Not using cursors results in repeated parses. If bind variables are not used, then there is hard parsing of all SQL statements. This has an order of magnitude impact in performance, and it is totally unscalable. Use cursors with bind variables that open the cursor and re-execute it many times. Be suspicious of applications generating dynamic SQL.

3. Getting Database I/O Wrong

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

4. Redo Log Setup Problems

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

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

   This is particularly common on `INSERT`-heavy applications, in applications that have raised the block size to 8K or 16K, or in applications with large numbers of active users and few rollback segments.

6. Long Full Table Scans

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

7. In Disk Sorting

   In disk sorts for online operations could indicate poor transaction design, missing indexes, or poor SQL optimization. Disk sorts, by nature, are I/O-intensive and unscalable.

8. High Amounts of Recursive (SYS) SQL

   Large amounts of recursive SQL executed by SYS could indicate space management activities, such as extent allocations, taking place. This is unscalable and impacts user response time. Recursive SQL executed under another user ID is probably SQL and PL/SQL, and this is not a problem.

9. Schema Errors and Optimizer Problems

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

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

10. Use of Nonstandard Initialization Parameters

    These might have been implemented based on poor advice or incorrect assumptions. In particular, parameters associated with `SPIN_COUNT` on latches and undocumented optimizer features can cause a great deal of problems that can require considerable investigation.

Performance Characteristics of Hardware Configurations

Again, today's systems are so different and complex that hard and fast rules for performance analysis cannot be made. However, this section provides some of the numbers that you should consider.

1. Disk characteristics:
   - Size 512M - 36Gig
   - Seek 5 - 10ms
   - Transfer 5 - 10ms
   - Thoughput 20 - 40 I/O seconds per disk
   - Controller throughput at 1750 I/Os per second
2. Speed reading from memory should be 1 - 10 microseconds.
3. Point-to-point network latencies should be 1 - 25ms.
4. Busy systems: (worst case)
   - Operational systems - 60%usr 40%sys
   - Decision support systems - 90%usr 10%sys

3

Emergency Performance Techniques

This chapter provides techniques for dealing with performance emergencies. Hopefully, you have had the opportunity to read the first two chapters of this book, where a detailed methodology is defined for establishing and improving application performance. However, in an emergency situation, a component of the system has changed to transform it from a reliable, predictable system to one that is unpredictable and not satisfying user requests.

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

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

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

The Emergency Performance Method is as follows:

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

   - User feedback on how the system is underperforming. Is the problem throughput or response time?
   - Ask the question, "What has changed since we last had good performance?" This answer can give clues to the problem; however, getting unbiased answers in an escalated situation can be difficult.

2. Sanity-check the hardware utilization of all components of the application system. Check where the highest CPU utilization is, and check the disk, memory usage, and network performance on all the system components. This quick process identifies which tier is causing the problem. If the problem is in the application, then shift analysis to application debugging. Otherwise, move on to database server analysis.

3. Determine if the database server is constrained on CPU or if it is spending time waiting on wait events. If the database server is CPU-constrained, then investigate the following:

   - Sessions that are consuming large amounts of CPU at the operating system level
   - Sessions or statements that perform many buffer gets at the database level (check `V$SESSTAT`, `V$SQL`)
   - Execution plan changes causing sub-optimal SQL execution (these can be difficult to locate)
   - Incorrect setting of initialization parameters
   - Algorithmic issues as a result of code changes or upgrades of all components

   If the database sessions are waiting on events, then follow the wait events listed in `V$SESSION_WAIT` to determine what is causing serialization. In cases of massive contention for the library cache, it might not be possible to logon or submit SQL to the database. In this case, use historical data to determine why there is suddenly contention on this latch. If most waits are for I/O, then sample the SQL being run by the sessions that are performing all of the I/Os.

4. Apply emergency action to stabilize the system. This could involve actions that take parts of the application off-line or restrict the workload that can be applied

to the system. It could also involve a system restart or the termination of job in process. These naturally have service level implications.

5. Validate that the system is stable. Having made changes and restrictions to the system, validate that the system is now stable, and collect a reference set of statistics for the database. Now follow the rigorous performance method described earlier in this book to bring back all functionality and users to the system. This process may require significant application re-engineering before it is complete.

   See Also: Chapter 2, "Monitoring and Improving Application Performance" for detailed information on the performance improvement method and a list of the most common mistakes made with Oracle

Index